

Operating System/2™ Programming Tools and Information Version 1.2

Programming Overview

64F3830

Operating System/2™ Programming Tools and Information Version 1.2

Programming Overview

First Edition (September 1989)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Note to US Government users – Documentation related to Restricted Rights – Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

A form for reader's comments appears at the back of this publication. If this form has been removed, address your comments to:

International Business Machines Corporation
Department 25X (Internal Zip 3613)
P. O. Box 1328B
Boca Raton, FL 33429-9960

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "© (your company name) (year) All Rights Reserved."

© Copyright International Business Machines Corporation 1989. All rights reserved.
Note to US Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Special Notices

The following names, used in this publication, are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries:

IBM
C/2
COBOL/2
FORTRAN/2
Macro Assembler/2
Operating System/2
OS/2
Presentation Manager
Systems Application Architecture.

The following names, used in this publication, are trademarks or registered trademarks of other companies:

CodeView Intel
Microsoft Corporation

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license enquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Armonk, NY 10504.

© Copyright IBM Corp. 1989

Preface

About This Book

The Operating System/2 Version 1.2 (OS/2) *Programming Overview* introduces programming concepts for OS/2 applications. This *Programming Overview* also introduces and describes the set of books, tools, programming aids, and sample applications that make up the OS/2 Programming Tools and Information package (tools package). **Read this *Programming Overview* first.**

Prerequisite Knowledge

The *Programming Overview* and the rest of the books in the programming library are intended for professional application developers knowledgeable in at least one programming language in which OS/2 applications can be written, including:

- C/2 Version 1.1
- Macro Assembler/2 Version 1.0
- COBOL/2 Version 1.0
- FORTRAN/2 Version 1.02
- Pascal Compiler/2 Version 1.0
- BASIC Compiler/2 Version 1.0

The information in the programming library assumes you are new to programming with the OS/2 control program, the Presentation Manager, and the Dialog Manager. You should understand the OS/2 services available to users. If you need help, consult the following information in the OS/2 product library:

- *Getting Started*
- *Using Advanced Features*
- *Product Information*

The following information products can be purchased individually:

- *Command Reference*
- *Service Coordinator's Guide*

Keyboards and Code Pages can be ordered separately at no cost.

Related Information

Programming Tools and Information

- *Installation*
- *Programming Overview*
- *Building Programs*
- *Programming Guide*
- *Control Program Programming Reference*
- *Presentation Manager Programming Reference Volume 1*
- *Presentation Manager Programming Reference Volume 2*
- *I/O Subsystems and Device Support Volume 1: Device Drivers*
- *I/O Subsystems and Device Support Volume 2: Presentation Driver Interface*
- *Dialog Manager Guide and Reference*
- *Dialog Tag Language Guide and Reference*
- *Dialog Manager and Dialog Tag Language Reference Summary*
- *Presentation Manager C/2 Bindings Reference*
- *Presentation Manager Macro Assembler/2 Bindings Reference*
- *Presentation Manager COBOL/2 Bindings Reference*
- *Presentation Manager FORTRAN/2 Bindings Reference*
- *Systems Application Architecture: Common User Access: Advanced Interface Design Guide*

Systems Application Architecture Library

- *An Overview*
- *Writing Applications: A Design Guide*
- *Common User Access: Basic Interface Design Guide*
- *Common Programming Interface: C Reference*
- *Common Programming Interface: COBOL Reference.*
- *Common Programming Interface: FORTRAN Reference*
- *Common Programming Interface: Dialog Reference*
- *Common Programming Interface: Presentation Reference*

© Copyright IBM Corp. 1989

OS/2 and Related Product Libraries

OS/2 Product

IBM Operating System/2
Standard Edition
Version 1.2

Getting Started
Using Advanced Features
Product Information

6024926 3.5″ diskette
6024930 5.25″ diskette

OS/2 Programming Tools and Information

IBM Operating System/2
Version 1.2

Installation
Programming Overview
Programming Guide
Building Programs
Dialog Tag Language Guide and Reference
Dialog Manager Guide and Reference
Dialog Manager and Dialog Tag Language Reference Summary
Programming Reference (3 books)
Bindings Reference for Presentation Manager (4 books for COBOL/2, FORTRAN/2, C/2, and Macro Assembler/2)
I/O Subsystems and Device Support (2 books)
Systems Application Architecture Common User Access: Advanced Interface Design Guide

6024929

Separate Order (no charge)

Keyboards and Code Pages

6280345

Available Separately

Command Reference

6024928

Service Coordinator's Guide

15F2214

Programming Languages

IBM Basic Compiler/2
Version 1.0 6280179

IBM Macro Assembler/2
Version 1.0 6280181

IBM Pascal Compiler/2
Version 1.0 6280183

IBM FORTRAN/2
Version 1.02 6280185

IBM COBOL/2
Version 1.0 6280207

IBM C/2
Version 1.1 6280284

Systems Application Architecture

An Overview

GC26-4341

Writing Applications: A Design Guide

SC26-4362

Common User Access: Advanced Interface Design Guide
SC26-4582

Common User Access: Basic Interface Design Guide
SC26-4583

Common Programming Interface

C Reference - Level 2

SC09-1308

COBOL Reference

SC26-4354

Dialog Reference

SC26-4356

FORTRAN Reference

SC26-4357

Procedures Language Reference
SC26-4358

Presentation Reference

SC26-4359

Contents

Chapter 1. Libraries and Tools for the OS/2 Programming Environments 1-1
OS/2 Version 1.2: New and Enhanced Functions 1-1
Presentation Manager 1-1
Dialog Manager 1-1
OS/2 Version 1.2: New and Enhanced Programming Library 1-1
Online Programming Reference 1-2
Programming Library 1-2
Systems Application Architecture Library . 1-3
Programming Aids and Sample Programs . . 1-4
Resource Editors 1-4
Compilers 1-4
Utilities 1-4
Source Header Files 1-5
Sample Programs 1-5

Chapter 2. OS/2 Programming Concepts . . . 2-1
The OS/2 Operating Environment 2-1
The Application Programming Interface 2-2
Structuring the Application for the Multitasking Environment 2-3
Sessions 2-4
Processes 2-4
Threads 2-4
Dispatching Priority 2-5
Communicating Among Processes 2-5
Flags 2-5
Pipes 2-5
Queues 2-6
Shared Memory 2-6
Semaphores 2-6
Managing File I/O 2-7
Managing I/O for Handle-Based Devices . . . 2-8
Managing Subsystem I/O: Video, Keyboard, and Mouse 2-8
Replacing a Subsystem 2-9
Device Drivers 2-9
Dynamic Linking 2-10
Dynamic Linking at Load Time 2-10
Loading on Demand 2-11
Dynamic Linking at Run Time 2-11
Managing Memory 2-11
Programming Considerations for the 80386 Microprocessor 2-12
Text Messages 2-12
System Limits 2-12

Chapter 3. Presentation Manager Concepts . 3-1
Systems Application Architecture 3-1
The Windowing Environment 3-2
Creating and Classifying Windows 3-2
Defining Window Relationships 3-2
Structuring the Application 3-3
Handling Mouse and Keyboard Input 3-3
Processing Messages 3-4
Providing the User Interface 3-5
Standard Window 3-5
Dialog Box 3-6
Message Box 3-6
Developing a Help Interface 3-7
Handling Application Resources 3-7
Resource Editors 3-8
Managing Graphic Objects 3-8
Creating and Storing Graphic Pictures . . . 3-8
Producing Character Graphics 3-8
Assembling Complex Pictures 3-9
Clipping 3-9
Transforming Graphic Objects 3-9
Handling Graphic Output 3-9
Graphics Presentation Spaces 3-9
Device Contexts 3-10
Metafiles 3-10
Bit Maps 3-10
Handling Alphanumeric Output 3-10
Producing Hard-Copy Output 3-10
Exchanging Data Among Applications 3-11
User-Generated Data Exchange 3-11
Application-Generated Data Exchange . . 3-12
Presentation Drivers 3-12

Chapter 4. Dialog Manager Concepts 4-1
Designing the Application Panel 4-1
Defining Dialog Elements 4-2
Manipulating Dialog Services 4-3
Managing Dialog Data 4-4
Associating Data with Dialog Elements . . . 4-4
Using Variables and Pools 4-4
Using System Variables 4-4
Translating and Validating User Input 4-4
Developing a Dialog Manager Application . . 4-5
Deciding When to Use the Dialog Manager . . 4-5
Conforming to Customer User Access . . . 4-5
Understanding Differences in Programming Environments 4-5
Understanding Differences in Application Structure 4-6
Creating and Displaying Windows, Panels, and Other Objects 4-6
Providing Help in Presentation Manager and Dialog Manager Applications 4-6
Using Presentation Manager Functions in Dialog Manager Applications 4-7

© Copyright IBM Corp. 1989

Providing Support for Language Translation . 4-7
Double-Byte Character Set Support ... 4-8
Providing Dialog Manager Code Page Support 4-8

Chapter 5. The Five Types of Supported Applications 5-1
Presentation Manager Application 5-1
Dialog Manager Application 5-1
Text Window Application 5-2
Full-Screen OS/2 Application 5-2
Full-Screen DOS Application 5-2
Family Applications 5-2
Comparing the Five Types of Applications .. 5-3
Programming Language Support 5-4
National Language Support 5-5

Appendix A. Programming Tools and Information Installation A-1
Installation Procedure A-1
The OS/2 Programming Tools Subdirectory Structure A-2
UNPACK Command A-3

Index X-1

Chapter 1. Libraries and Tools for the OS/2 Programming Environments

This *Programming Overview* introduces you to the programming concepts, books, sample programs, and programming aids that help you in developing applications that run on Operating System/2™[1] Version 1.2 (OS/2®[2]), the Presentation Manager™[1], and the Dialog Manager. After reading this book, you should know which type of supported application you want to write and what information and facilities are available to support your effort.

We recommend that you take some time to read this book before you begin programming.

OS/2 Version 1.2: New and Enhanced Functions

OS/2 Version 1.2 replaces OS/2 Version 1.1 as the primary operating system for 80286- and 80386-based systems. In addition to features such as concurrent processing, large memory addressability, graphics, windowing, and DOS compatibility, the OS/2 Version 1.2 control program supports larger files, long file names, extended attributes, faster access to files, and multiple file systems.

Presentation Manager

The Presentation Manager component of OS/2, introduced in OS/2 Version 1.1, allows software vendors to produce competitive, graphic-based applications that are similar and easy to use. Enhancements to the Presentation Manager include the following:

- Bindings for COBOL/2 and FORTRAN/2 programming languages
- Information Presentation Facility for developing and integrating help information into application programs
- Additional features to aid compliance with Systems Application Architecture™[1]
- Additional font support
- Additional function in the print spooler.

Dialog Manager

The Dialog Manager component of OS/2 Version 1.2 allows programmers in large enterprises to produce text-based applications quickly and easily. Applications written using the Dialog Manager interface are similar in appearance to ISPF/EZ-VU applications.

Using the Dialog Manager interface supports compliance with the Common User Access. If desired, Dialog Manager applications can be written to use the Presentation Manager graphic user interface.

OS/2 Version 1.2: New and Enhanced Programming Library

The OS/2 Version 1.1 programming documentation consisted of a Programmer's Toolkit library containing guidance information and a Technical Reference library containing reference information. The two libraries could be ordered separately using two part numbers. The OS/2 Version 1.2 programming library combines the OS/2 Version 1.1 libraries into one product with a single part number and contains new information to support the Dialog Manager and other enhancements to the OS/2 Version 1.2 product.

Some of the book titles from the OS/2 Version 1.1 programming library have been renamed for OS/2 Version 1.2 to be more descriptive. For example, the *I/O Subsystems and Device Drivers* books have been renamed *I/O Subsystems and Device Support Volume 1: Device Drivers*, and *I/O Subsystems and Device Support Volume 2: Presentation Driver Interface*.

Also, some of the information from Version 1.1 has been moved into different books in the Version 1.2

[1] Trademark of IBM Corporation

[2] Registered Trademark of IBM Corporation

© Copyright IBM Corp. 1989

library. Control Program function calls (Dos, Kbd, Mou, and Vio) and their associated C/2 and Macro Assembler/2 bindings have been relocated into the *Control Program Programming Reference*. The Presentation Manager function calls (Dev, Gpi, Pic, Prf, Spl, advanced Vio, and Win) are in *Presentation Manager Programming Reference Volume 1*; related reference information is in *Presentation Manager Programming Reference Volume 2*.

New books are:

- *Dialog Manager Guide and Reference*
- *Dialog Tag Language Guide and Reference*
- *Dialog Manager and Dialog Tag Language Reference Summary*
- *Presentation Manager FORTRAN/2 Bindings Reference*
- *Presentation Manager COBOL/2 Bindings Reference*
- *Systems Application Architecture: Common User Access: Advanced Interface Design Guide*

Online Programming Reference

New to OS/2 is the online Programming Reference that allows you to view Presentation Manager and control program function calls. Reference information such as language bindings, related calls, or example code is available when a call is specified. An introduction provides instruction on the use of the program.

The online *Programming Reference* duplicates information in these books:

- *Control Program Programming Reference*
- *Presentation Manager Programming Reference Volume 1*
- *Presentation Manager Programming Reference Volume 2*
- *Presentation Manager C/2 Bindings Reference*
- *Presentation Manager Macro Assembler/2 Bindings Reference*
- *Presentation Manager COBOL/2 Bindings Reference*
- *Presentation Manager FORTRAN/2 Bindings Reference*

Programming Library

Take some time to sort out the books in the programming library and determine which ones you need to use.

All OS/2 application developers should be familiar with this book, the *Programming Overview*, and with the *Systems Application Architecture: Common User Access: Advanced Interface Design Guide* (Design Guide). The *Programming Overview* provides general concepts and helps you make design decisions. The Design Guide provides guidelines for designing applications with a consistent user interface.

To your language library, add *Building Programs* to guide you in building your executable files and libraries. OS/2 applications can be written, with some restrictions, in the following programming languages:

- C/2[TM3] Version 1.1
- Macro Assembler/2[TM3] Version 1.0
- COBOL/2[TM3] Version 1.0
- FORTRAN/2[TM3] Version 1.02
- Pascal Compiler/2[TM3] Version 1.0
- BASIC Compiler/2[TM3] Version 1.0

Note that levels of support vary. See "Programming Language Support" on page 5-4 for details.

Developing a Dialog Manager application: Use the *Dialog Manager Guide and Reference* and the *Dialog Tag Language Guide and Reference*.

The *Dialog Manager Guide and Reference* provides guidance information supported with code fragments from the sample programs included in the tools package. The book describes how to use the services of the Dialog Manager.

The *Dialog Tag Language Guide and Reference* contains guide and reference information about the tag language used to interface with the Dialog Manager and help facilities. The book includes an alphabetic list of the all the tags, the syntax for each tag, a definition of each parameter, and examples of how to use the tags. The book also includes guidance and reference information for

[3] Trademark of IBM Corporation

using the utility that compiles the tagged source files, the Dialog Tag Language Compiler.

Developing a Presentation Manager application: Use the *Programming Guide* and use the online *Programming Reference*, or the *Presentation Manager Programming Reference Volume 1*, *Presentation Manager Programming Reference Volume 2*, and one of the language bindings references.

The *Programming Guide* helps you in developing the source code for your application by describing common programming techniques, sometimes using detailed walk-throughs of the sample programs included in the tools package. The *Programming Guide* explains how to write the five types of applications that run in the OS/2 operating environments. You can read the book sequentially but, more than likely, you will refer to relevant parts of the book as you develop your source code.

OS/2 allows Presentation Manager applications written in C/2 Version 1.1, Macro Assembler/2 Version 1.0, COBOL/2 Version 1.0, and FORTRAN/2 Version 1.02 to request services directly using Application Programming Interface (API) calls. The Programming Reference books contain comprehensive, alphabetical listings of all the OS/2 control program and Presentation Manager API calls used to request system services, including their input and output parameters, error return codes, data structures, and messages.

As you write your source code following programming techniques recommended in the *Programming Guide*, use the programming reference books as your reference to all the API calls. The programming reference books provide the syntax for each API call in a *metalanguage*, a format independent of language-specific calling conventions.

The *Presentation Manager C/2 Bindings Reference*, *Presentation Manager FORTRAN/2 Bindings Reference*, *Presentation Manager COBOL/2 Bindings Reference*, and *Presentation Manager Macro Assembler/2 Bindings Reference* books contain language-specific syntax, including parameters and data types, for each API call. Information in the bindings references is derived from the source header files described in "Source Header Files" on page 1-5.

Adding control program function calls: To add control program function calls in your application, use the *Programming Guide* and use the online *Programming Reference*, or the the *Control Program Programming Reference*. Guidance information for control program calls is primarily in the section in the *Programming Guide* on control program functions and alphanumeric text applications. Control program bindings for C/2 and Macro Assembler/2 are integrated into the *Control Program Programming Reference*.

Developing OS/2 device drivers, subsystems, and presentation drivers: The *I/O Subsystems and Device Support Volume 1: Device Drivers* and *I/O Subsystems and Device Support Volume 2: Presentation Driver Interface* books contain guide and reference information about the OS/2 device drivers, subsystems, and presentation drivers. The books guide you in using the I/O application interfaces. They also explain how to write device drivers, subsystems, and presentation drivers to replace or extend those available with the system.

Systems Application Architecture Library

OS/2 participates in Systems Application Architecture (SAA). The Presentation Manager implements the common programming interface (CPI) component of SAA, which facilitates program portability to other environments supported by SAA. The OS/2 user interface conforms to the Common User Access (CUA) component of SAA. The *Programming Guide* describes how to write your Presentation Manager application so that it conforms to SAA guidelines, making it similar in appearance and behavior to other SAA-conforming applications. The Dialog Manager implements the dialog interface component of SAA, which enables compliance with many CUA guidelines. See *Systems Application Architecture: Common User Access: Advanced Interface Design Guide* which describes CUA conventions.

The books in the SAA library are listed in "OS/2 and Related Product Libraries" on page vi. See your IBM marketing representative for ordering information.

Programming Aids and Sample Programs

The tools package contains the source code for many sample programs that demonstrate common programming techniques using the control program, Presentation Manager, and Dialog Manager calls. A sample printer presentation driver has been added. The sample programs are well documented with descriptive prologues and internal commentary.

The tools package also provides the Dialog Manager, the *Programming Reference*, resource editors, utilities, and programming aids such as header files, dynamic link library files, and sample module definition files to help you develop your application. Other utilities are available with the OS/2 product and in the C/2 compiler package.

An *Installation* card describes how to install the sample programs. The *Installation* card illustrates the subdirectory structure created following the installation procedure described in the card. For an additional copy of the installation instructions see Appendix A, "Programming Tools and Information Installation" on page A-1.

Resource Editors

The tools package contains three interactive editors for developing resources for Presentation Manager applications.

Dialog Box Editor
Supports interactive creation of dialog boxes and saves the definitions in a resource file.

Font Editor
Supports interactive editing of font files, saves the definitions in a font file, and includes the font file names in the application's resource file.

Icon Editor
Supports interactive creation of customized icons, pointers, and bit maps and saves the definitions in a resource file.

Compilers

The tools package contains three compilers for Presentation Manager and Dialog Manager applications.

Resource Compiler (RC)
Used in the Presentation Manager environment, the RC processes the text resource file to produce a binary file. The binary file is attached to the application's executable file or to a dynamic link library file to provide access to the resources.

Dialog Tag Language Compiler
Converts source dialog tag language into Dialog Tag Language elements for use in Dialog Manager applications.

Information Presentation Facility Compiler
Converts IPF Tag Language into an Information Presentation Facility library format.

Utilities

The following utilities help you to build and debug your application:

LINK
Produces executable files from object files for the Presentation Manager, OS/2, and DOS[TM4] environments. The Linker/2[TM4] Version 1.2 (LINK) is packaged with the Operating System/2 Version 1.2 product.

OS2STUB.EXE
Augments the STUB statement in the module definition file. Provided by OS/2, this stub replaces the default stub message displayed by the LINK utility.

IMPLIB
Creates an import library that resolves an application's external references to subroutines in dynamic link libraries.

BIND
Creates a DOS executable file from an OS/2 executable file.

MKMSGF
Converts an error, help, prompt, or general information text message file to binary format for display at run time.

[4] Trademark of IBM Corporation

MSGBIND
Binds a text message created with the MKMSGF utility as a data segment to the application's executable file.

CodeView™[5]
A symbolic debugger for applications written in C/2, Macro Assembler/2, Pascal Compiler/2, and BASIC Compiler/2. CodeView is packaged with C/2 Version 1.1 and can be ordered using a form in the OS/2 Version 1.2 product library.

MAKE
A compiling aid that searches for files changed since the last compile and recompiles only the changed files. MAKE is packaged with C/2 Version 1.1 and Macro Assembler/2 Version 1.0.

WINABLE
Converts most full-screen OS/2 applications so that they can run in a Presentation Manager window.

Source Header Files

Source C/2 and Macro Assembler/2 header files are provided in the tools package and described in the README.C and README.ASM files. The header files, OS2.H for C/2 and OS2.INC for Macro Assembler/2, contain declarations and equates used in the control program and Presentation Manager API calls. An include statement in your source code automatically calls a hierarchy of header files containing the code and parameters for the control program and Presentation Manager functions. Most of the files allow you to tailor which subsets of functions you want to include.

COBOL/2 include files have a .CIN extension, and FORTRAN/2 include files have a .FIN extension and are discussed in the README.CBL and README.FOR files. Note there is no support for the selective inclusion of Presentation Manager include files as there is for C/2 and Macro Assembler/2. There is no equivalent to the OS.H file and the underlying structure of include files that are supplied for C/2 and Macro Assembler/2.

Sample Programs

The tools package contains the C/2 and Macro Assembler/2 source code for many sample programs that demonstrate common programming techniques using the OS/2 and Presentation Manager API calls. You can compile and run the sample programs.

Most of the sample programs are written in C/2. Because there are many options available when you set up and install the C/2 compiler, we standardized a set of options and naming conventions to ensure that you can compile, link, and run the OS/2 and Presentation Manager sample programs. The following libraries are required:

SLIBCE.LIB
The small-model library

MLIBCE.LIB
The medium-model library

CLIBCE.LIB
The compact-model library

LLIBCE.LIB
The large-model library

LLIBCMT.LIB
The multi-threaded library.

Specify certain options when you build the C/2 libraries. When prompted for the operating system, you must specify OS/2, although others can be specified. When prompted for floating point options, specify the emulator option. When prompted for the names of default libraries, specify the OS/2 default libraries. When asked if you want the combined set of libraries, specify that you do.

Command and Make Files: To ensure that you are able to compile, link, and run the sample programs, a make (.MAK) file is provided for each program. The make file contains a list of both compiler and link commands used by the MAKE utility to run each sample program. The command file provides the names of libraries required to run each sample program. The make file is called by the command (.CMD) file.

Sample programs written in C/2 language, assume you are using the MAKE utility provided with IBM

[5] Trademark of Microsoft Corporation

C/2 Version 1.1. Similarly, sample programs written in Macro Assembler/2, assume use of the MAKE utility provided with IBM Macro Assembler/2 Version 1.0. The MAKE utility is not interchangeable, which means you cannot call the MAKE utility with C/2 to compile a Macro Assembler/2 sample program, or call the MAKE utility with Macro Assembler/2 to compile a C sample program. However, you can edit the command file affiliated with the sample program and change the format on how the MAKE utility is called. Instructions are provided in the command file.

Figure 1-1 illustrates the VIOCBAT.CMD command file for the VIOSAMPC.C sample program. Figure 1-2 illustrates the VIOCBAT.MAK make file for the VIOSAMPC.C sample program.

```
************** Keyboard and VIO C Sample Program Batch File ***************

This is the batch file containing the commands to compile, link, and run
the Keyboard and VIO C Sample Program.  ** Note:  A call is made to the
batch file CSAMPATH.CMD which sets the environment for this process.

Required Programs:
 IBM C/2 (version 1.1 or later)
 IBM Linker/2 (version 1.1 or later)

Required Files:

 **NOTE** : See Readme.C for a list and description of Required
            SYSTEM files. This file is located in the \toolkt12\c
            directory.

  viocbat.mak is the makefile for the make utility
  viosampc.c is the source code for this Sample Program.
  mlibce.lib is the medium model protect/standard combined C library.
  doscalls.lib is the library the linker will search to resolve external
  Dynamic Link references for System Calls.

*****************************************************************************


%1\toolkt12\c\samples\bse\csampath.cmd
cd viocbat

*****************************************************************************
* the following call is for the IBM MASM/2 1.0 make utility                 *
*****************************************************************************
* make viocbat.mak                                                          *
*****************************************************************************
* the following call is for the IBM C/2 1.10 make utility                   *
*****************************************************************************
 make /f viocbat.mak viosampc.exe
 You are now ready to run the Keyboard and VIO Sample you just built: pause

viosampc
```

Figure 1-1. Example of a Command File

```
#  Compiler Switches:
#   /Zl - Suppress default library selection.
#   /Zp - Pack structure members - OS/2 API calls expect packed structures.
#   /Ze - Enable special keywords.
#   /G0 - Enable 8086 / 8088 instruction generation.
#   /Gs - Remove stack probes - Use only on fully debugged program.
#   /AM - Using the Medium memory model.

viosampc.obj: viosampc.c
        cl /W3 /c /Zl /Zp /Ze /G0 /Gs /AM viosampc.c

viosampc.exe: viosampc.obj
        link viosampc,,,mlibce.lib doscalls.lib;
```

Figure 1-2. Example of a Make File

Following are the sample programs written in C/2 that demonstrate programming techniques using OS/2. Comparable sample programs are available in Macro Assembler/2.

CRERRCB.C
Demonstrates the handling of a critical error by generating a pop-up window containing an error message.

D_INITC.C
Demonstrates an initializer routine for a dynamic link library.

DLNK1C.C
Creates a subroutine that will be incorporated into a dynamic link library as a pre-load module.

DLNK2C.C
Demonstrates how to add a subroutine to a dynamic link library and load it when called by the program.

DLNK3C.C
Demonstrates how to add a subroutine to a dynamic link library and load the subroutine by explicitly calling it at run time.

DYNLINKC.C
Demonstrates three ways in which a program can call a subroutine in a dynamic link library. One subroutine is pre-loaded by the loader. Another subroutine is loaded when called by the program. The third subroutine is explicitly loaded, called, and freed by the program at run time.

HELLOCB.C
Displays a text message on the screen.

MEMORYCB.C
Demonstrates how to allocate and deallocate memory.

MOUSEC.C
Demonstrates how to receive mouse input and write it to the screen.

VIOSAMPC.C
Demonstrates how to receive keyboard input and write it to the screen.

WTC.C
Demonstrates how to display the date in two country-specific formats.

The following OS/2 sample programs are written in Macro Assembler/2 only:

MONITOR.ASM
Demonstrates how to structure a device monitor and how the device monitor filters data passed between it and the device driver.

PROCESS.ASM, PROG1.ASM, PROG2.ASM
Demonstrate how to create multiple processes that share memory, create child processes, and pass data using queues and shared memory (PROG1) and pipes (PROG2).

THREADB.ASM
Demonstrates how to create two threads that write data to the screen.

DEMODB.ASM
Demonstrates how to structure a device driver, including the strategy routine and the timer handler. This sample program also demonstrates how to send data to a device monitor. The data received from the device monitor is placed in a queue managed by the timer handler.

IOPL2B.ASM
Demonstrates how to call a subroutine that executes with I/O privilege.

PROMODE.ASM
Demonstrates how to execute a family program in the OS/2 environment.

REALMODE.ASM
Demonstrates how to execute a family program in the DOS environment.

The sample programs that demonstrate common programming techniques in the Presentation Manager environment are written in C/2 except HELLOM, which is written in Macro Assembler/2. The Presentation Manager sample programs include:

AVIOSAMP
Demonstrates how to display alphanumeric text on the screen and then print it. A window sub-classing procedure is implemented to allow the program to intervene in sizing a window. The sample program also demonstrates how to create and manipulate a child window and how to allocate and free memory segments.

BMAP
Demonstrates how to paint a window using data stored in a bit map in memory.

CLIPBRD
Demonstrates how to cut and paste data from one window to another using a bit map.

DIALOG1
Demonstrates how to implement and display a dialog box in a standard window. Comparable sample programs are written in COBOL/2 and FORTRAN/2.

The *Programming Guide* provides a detailed walk-through of this sample program.

DIALOG2
Demonstrates how to implement entry fields, radio buttons, check boxes, list boxes, push buttons, and message boxes. Comparable sample programs are written in COBOL/2 and FORTRAN/2.

EALIST
Provides source files written in the C/2 language that allows you to view and modify the extended attributes mechanism.

FONTTEST
Demonstrates how to load dynamic link libraries containing fonts, query the system-supplied fonts, paint a window, display an hour-glass-shaped pointer, and implement a dialog box. This sample program is an amplification of the TEMPLATE sample program.

GRAPHIC1
Demonstrates how to draw, retain (store) and redraw graphic data from a metafile onto the screen and then print it. A thread is created with a message queue to monitor asynchronous user activities.

GRAPHIC2
Demonstrates how to draw non-retained graphic data from a metafile onto the screen and print it using a print spooler. Two threads are created with a message queue to monitor asynchronous user activities.

HELLO1
Demonstrates the basic structure common to all Presentation Manager programs.

The *Programming Guide* provides a detailed walk-through of this sample program.

HELLO2
Demonstrates how to create and display a standard window with an action bar containing a menu and a pull-down menu.

The *Programming Guide* provides a detailed walk-through of this sample program.

HELLOM
Demonstrates how to create and display a standard window with an action bar containing a menu and a pull-down menu. This sample program is written in Macro Assembler/2.

HELP1
Demonstrates how to call the Information Presentation Facility help hook, to create a help instance and associate the instance with the active application window.

Also demonstrates how to display information in the help text window, in response to a user request for help.

The *Programming Guide* provides a detailed walk-through of this sample program.

IMAGE

Demonstrates how to produce a graphic picture on the screen that responds to user scrolling. A window sub-classing procedure is implemented to allow the program to intervene in sizing the window.

This sample program also demonstrates how to create and manipulate a dialog box, how to set global color attributes, and how to create and manipulate a frame window and controls.

PMPRT

Demonstrates programming techniques written in C/2, Macro Assembler/2, COBOL/2 and FORTRAN/2 for coding a printer presentation driver.

TYPETEXT

Demonstrates how character input is handled and echoed to a window using messages. Font sizes are queried to ensure device independence.

TEMPLATE

Demonstrates the structure common to all Presentation Manager applications. This sample program shows how to structure an application having more than one source file. It includes an initialization file which is used and then discarded, the resident code, and the non-resident code that is loaded only when needed.

Demonstrates how to create a standard window and load resources from a resource file, handle an error, create a dialog box containing a listbox and button controls, and display a message box.

The following sample program written in C/2, as well as COBOL/2, demonstrates programming techniques in the Dialog Manager environment:

DMDEMO

Provides application panels and help panels defined using Dialog Tag Language and program source code required to run a sample Dialog Manager application.

Chapter 2. OS/2 Programming Concepts

This chapter describes the programming concepts that will help you plan and design your application, including:

- The OS/2 operating environment
- The Application Programming Interface (API)
- Structuring the application for the multitasking environment
- Communicating among processes
- Managing file I/O
- Managing I/O for handle-based devices
- Managing subsystem I/O: video, keyboard, and mouse
- Device drivers
- Dynamic linking
- Managing memory
- Text messages
- System limits.

This chapter describes the OS/2 facilities available to most applications. See the *Programming Guide* for information about the explicit services available to each of the types of applications supported by OS/2.

Chapter 3, "Presentation Manager Concepts," describes the concepts and interfaces relevant to the Presentation Manager environment.

Chapter 4, "Dialog Manager Concepts," describes the concepts and services provided by a Dialog Manager environment and notes the differences between a Dialog Manager and Presentation Management application.

The OS/2 Operating Environment

OS/2 was developed to support the Personal Computer® and the Personal System/2® (PS/2®) families of products.[1] They include the PS/2 Models 30 (286), 50, 50Z, 60, 70, and 80. They also include the Personal Computer AT®[1] and the Personal Computer XT™ (Model 286).[2] These hardware systems are driven by the Intel 80286 or 80386 central processing unit (CPU). A minimum system configuration of 2.5 MB of memory and one fixed disk drive is required to run an OS/2 system.

OS/2 takes advantage of the virtual addressing mode of the processor, called *protected mode*, while retaining a DOS execution environment that uses the processor's real addressing mode, called *real mode*. The real mode is compatible with that of the 8088 processor and, with some exceptions, allows existing DOS applications to run in the OS/2 environment. In addition, an application can be written to run in both the DOS and the OS/2 environments. See Chapter 5, "The Five Types of Supported Applications" for more information.

The real mode of the processor limits the amount of addressable physical memory to 1MB. The protected mode supports up to 16MB of addressable memory.

Using *memory overcommitment*, OS/2 enables a single application, or all applications, to use a virtual address space. OS/2 implements *virtual memory* by writing the least frequently used memory segments to a temporary file on the disk and bringing the segments back into physical memory only as they are required. This is called *segment swapping*.

[1] Registered Trademark of IBM Corporation

[2] Trademark of IBM Corporation

© Copyright IBM Corp. 1989

Figure 2-1 illustrates the OS/2 memory model.

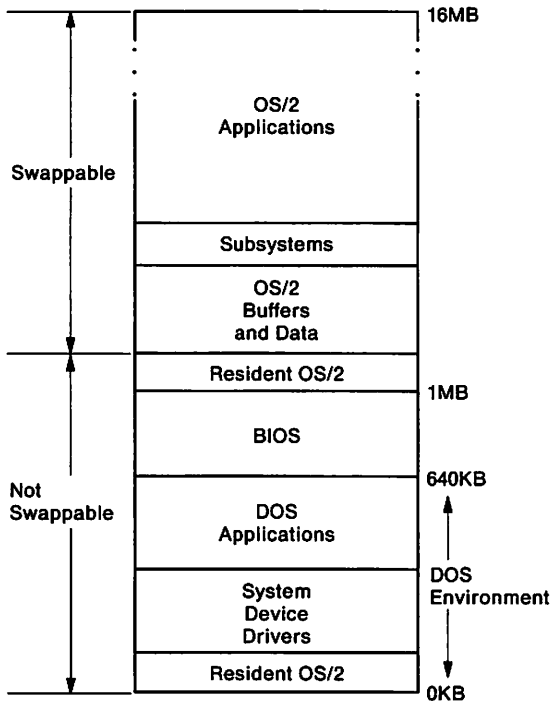


Figure 2-1. OS/2 Memory Model

In the OS/2 environment, the application sees a collection of segments, each no larger than 64KB, in place of a linear address space. In the DOS environment, the segment register contains the first 16 bits where a segment begins in physical memory.

In the OS/2 environment, the value placed in a segment register is called a *selector*. The selector identifies a segment in memory using an index into one of the *segment tables* maintained by OS/2. There are two tables, a *Local Descriptor Table* (LDT), which contains the local address space assigned to each application, and the *Global Descriptor Table* (GDT), which defines the system address space. Entries in the tables are called *descriptors* and are managed by OS/2. Descriptors contain information about the segments they describe, including the address at which the segment begins, its size, and its access privileges.

Protection for each application is implemented by way of the descriptors. Each application has its own address space defined in the LDT, and an application's memory segments are always addressed by way of the LDT. See "Managing Memory" on page 2-11 for a description of the memory management services available to the application.

In addition to protection in memory, applications in the OS/2 environment are isolated from one another and from the system by having limited use of I/O instructions that disable interrupts.

Figure 2-2 illustrates the OS/2 privilege model.

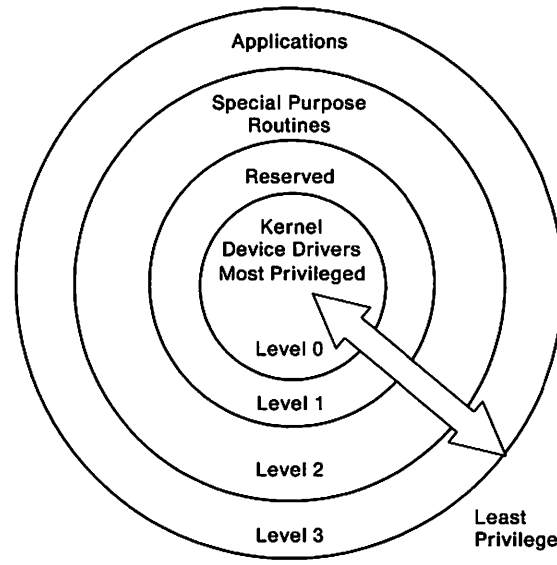


Figure 2-2. OS/2 Privilege Model

Applications run at privilege level 3, the least privileged level. Special purpose routines requiring I/O privilege run at privilege level 2. Privilege level 1 is reserved. The kernel and device drivers run at privilege level 0, which is the most privileged level. "Device Drivers" on page 2-9 explains the I/O privilege model in more detail.

The Application Programming Interface

More than 700 application services define the capabilities of OS/2 and the Presentation Manager. The application requests the OS/2 services using API calls. The API calls begin with prefixes that help you identify related subsets of services. They include the following:

Dev The services supporting presentation drivers

Dos The system services supporting memory management, process control, and handle-based file I/O

Gpi The services supporting graphics

Kbd The services supporting keyboard input

Mou The services supporting mouse input

Spl The services supporting hard copy spooling.

Vio The services supporting I/O to the video device

Win The services supporting windows

From the application's viewpoint, the code for each API call looks like a subroutine that can be called by name using an inter-segment far call. The subroutine that implements the API call is not part of the application's executable file. Relocation pointers to the subroutines are contained in a *dynamic link library*, which is described in "Dynamic Linking" on page 2-10. The loader fills in the addresses of the subroutines and fixes up the executable file so that it is ready to execute.

Before making an API call, the application pushes parameters, which may contain many types of data objects, in sequential order onto an execution stack. A subroutine called by a non-Presentation Manager application is responsible for removing data from the stack and setting the error codes in the AX register before returning control to the calling application. By contrast, a Presentation Manager application is returned data immediately in the AX or AX and DX registers if the result of the call is a double word.

There are advantages to the call-return method of accessing the system services:

- The application can call functions directly, eliminating the need for a DOS-style function router.
- The application's executable file is smaller than it would be if it contained all the code to implement the API calls.
- The API can be changed easily without affecting existing applications.

The implication, of course, is that the API can be extended with a new set of API functions. In fact, whether the kernel controls a function or whether it is contained in a dynamic link library, the mechanics of the API are the same. Following certain conventions, a developer can extend or replace an I/O *subsystem*. See "Managing Subsystem I/O: Video, Keyboard, and Mouse" on page 2-8 for more information.

The *Control Program Programming Reference*, *Presentation Manager Programming Reference Volume 1*, and *Presentation Manager Programming Reference Volume 2* contain a comprehensive, alphabetical listing of all the API calls, including their input and output parameters, error return codes, data structures, and messages.

Structuring the Application for the Multitasking Environment

The OS/2 environment allows multiple, complex applications to co-exist. Each application has its own virtual address space in memory. An application's address space is protected from disruption by other applications.

In a multiple-application environment, users can increase their productivity as they make efficient use of their computers. For example, data can be accessed and calculated for use by a spreadsheet application running in the background, while the user continues to type in data using a text editor application running in the foreground. Time-consuming activities such as compiling an application, obtaining data from a communications link, or printing a file can take place in the background.

Although all the applications appear to the user to be running simultaneously, the processor actually can perform only one task at a time. The processor often must wait for input or output from a relatively slow device, such as the keyboard or disk.

An application often has to perform multiple tasks that can be synchronized. To make efficient use of the processor, an application can be structured to allow one piece of work to execute while another is waiting for something to happen. The application can be written so that it can execute concurrently with other applications of which it has no knowledge.

OS/2 manages the concurrent processing of applications by making certain that the computer's resources are allocated properly to prevent one application from interfering with another one. However, to share the OS/2 environment, the application must plan and synchronize execution of its routines.

To co-exist harmoniously in the *multitasking* environment, each application must be able to communicate with other applications, synchronize its activities, and serialize its use of resources. It must be able to start, stop, and control its units of execution.

OS/2 helps to control the multitasking environment on three levels. The three-tier hierarchy addresses both the user's need to control concurrent execution and the developer's need to design applications whose activities can be performed concurrently.

Figure 2-3 illustrates the hierarchical relationships of the multitasking elements.

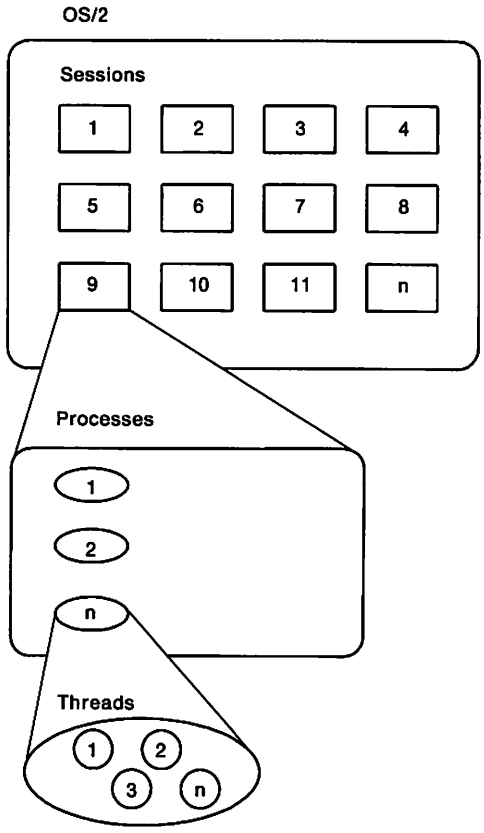


Figure 2-3. Multitasking Hierarchy

Sessions

A *session*, at the highest level of the OS/2 multitasking hierarchy, represents a logically separate unit of screen, keyboard, and mouse and their related processes. The logical devices are mapped to physical devices when the user selects a session by bringing it to the foreground to perform I/O. Generally speaking, each application runs in its own session. However, more than one application in different Presentation Manager sessions can share the physical screen, keyboard and mouse.

Sessions can be arranged in a hierarchy of parent and child descendancy. By default, a parent session can make a child session available for the user to select (start) from the Start Programs window in the Presentation Manager user shell. Or, a session can start another independent session. In this case, the starting session cannot control whether the user can start the second session. If a session is running in the foreground, it can bring any of its child sessions to the foreground. A parent session can end its own child sessions but cannot end their descendants. But, if a child session is ended, OS/2 automatically ends the descendant sessions.

Processes

A *process* is the logical unit of resources, including memory, files, pipes, queues, system semaphores, device monitors, disk files, and devices, allocated to run an application. Each session contains at least one process.

One process can create other processes that are arranged in a hierarchy of parent and child descendancy similar to that of sessions. A *child process* created by a *parent process* inherits access to certain resources owned by the parent process, including handles to files, pipes, and devices. The parent process retains control of its child processes and any other processes called in the line of descendancy. A process can control the execution of its child processes. A process can end itself and its children.

A process can give control to a set of routines to execute when the process ends. The routines, called *exit lists*, free allocated resources and perform "general housekeeping" after a normal or abnormal end.

Threads

The dispatchable unit of execution is called a *thread*. Each process has at least one thread. A thread does not own system resources but shares the resources owned by the process that created it. While a process is a static entity, a thread is dynamic. A running application is defined by its threads' execution stack, register values, and dispatch state (executing or waiting to execute). Threads are not organized hierarchically. Each thread within a process is a peer with the other threads within the process.

Using the API calls, a thread can start other threads within a process. A thread also can suspend and resume execution of other threads

temporarily to block them from using critical resources. Although all threads in a process are considered to be peers, if the first thread in a process ends, all other threads it created end, also. But, as each thread finishes executing, it is good programming practice to issue an API call to end itself. See "Communicating Among Processes" for more information.

Dispatching Priority

Unless explicitly defined using an API call, the system assigns each thread a dispatching priority. Processor time is allocated to only one thread at a time. OS/2 uses *time-slicing* to ensure that threads of equal priority are given equal chances to execute. OS/2 can preempt a thread when its time slice expires if a thread with a higher or equal priority is ready to execute. If not defined otherwise, the default minimum time-slice is 250 milliseconds. OS/2 timer services, accessed through the API, support all timer-related activities, including synchronizing the activities of threads.

When a thread is created, its priority class and priority level within the class are the same as that of its parent process. The API gives the application the ability to query or change a thread's priority class and level. The priority classes are *time-critical*, *fixed-high*, *regular*, and *idle-time*:

- Time-critical threads, such as those handling communications or real-time applications, require immediate execution. OS/2 assigns a time-critical thread to any of 32 priority levels, with round-robin dispatching from within the priority level.
- OS/2 assigns a fixed-high thread to any of 32 priority levels and dispatches it when no time-critical threads are waiting to execute.
- OS/2 assigns a regular thread to any of 32 priority levels and dispatches it based on the thread's display status (background or foreground), recent I/O activity, and processor use. Most threads are classified with regular priority.
- OS/2 assigns an idle-time thread to any of 32 priority levels and dispatches it when no regular, fixed-high, or time-critical threads are waiting to execute.

If the application does not use any time-critical devices, it should not be overly concerned with setting priority. Its threads will execute automatically with regular priority. (Fixed disks, diskettes, and printers generally are not time-critical.) However, knowing that the application will be running with others, you may be tempted to use priority level 32 to be sure it has its share of resources. Avoid assigning unnecessarily high priority levels.

Communicating Among Processes

In a multitasking environment, processes and threads need to communicate with one another to synchronize events and to control access to shared resources. OS/2 provides a set of interprocess communication (IPC) protocols for such purposes.

Flags

Flags are used when one process needs information about external events taking place in another process or in the system. If a process wishes to receive information about asynchronous events taking place in another process, the first process (the receiving process) creates a signal handler using the API. The second process (the sending process) uses an API call to signal, or *flag*, the first process that an event has occurred. The receiving process can synchronize its actions to respond to the event.

Pipes

A *pipe* is a fixed-length, circular buffer in memory. A pipe is identified by a handle. Because a pipe represents a serial character stream, it can be accessed like a file by processes that are cooperating closely. The obvious benefit is that data output from one application can be redirected through a pipe as input to another application.

Figure 2-4 on page 2-6 illustrates how data is transferred through a pipe.

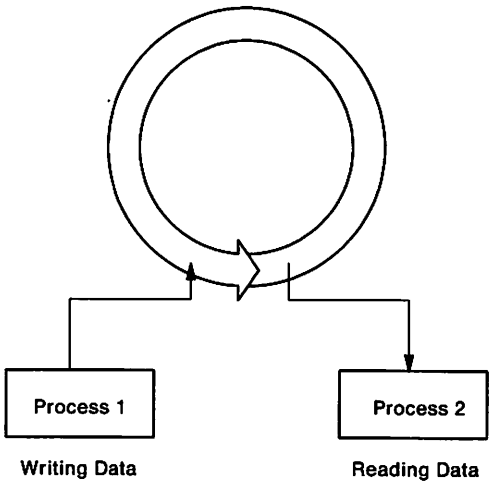


Figure 2-4. Piping Data

Another type of pipe, called a *named pipe*, provides two-way communication among unrelated processes, either locally or remotely. Generally, one process is a server process and the others are client processes. The server process creates the pipe and waits for a client process to open it. When the client requests the data in the pipe, the server process responds by sending it. "Managing File I/O" on page 2-7 for more details about how to use pipes.

Queues

A *queue* allows each element in a data stream to be addressed individually. A queue allows data elements to be accessed in first-in-first-out (FIFO) or last-in-first-out (LIFO) sequences. Or, the application can define the priority of a queue element. Although multiple processes can write to a queue, only the process that created it can read it. Individual elements in a queue can be examined without removing them.

A packet of data called a *message* can be sent to a queue by one thread in a process and retrieved, when required, by another thread in the process. The exchange does not need to be synchronized. The message is not copied to the queue but passed in a shared segment.

Shared Memory

Using *shared memory*, OS/2 enables two processes or all processes in the system to share a single memory segment.

To allow two processes to share memory, the first process allocates a memory segment and designates that it is to be shared. The process then issues a call to notify the system that a second process can share the memory. OS/2 returns the selector which references the memory segment. The first process then passes the selector to the second process by way of a queue.

If all processes in the system need to share a memory segment, the allocating process gives the segment a name. Using another means of interprocess communication, such as a queue or a pipe, the name is passed to other processes. The other processes then request access to the segment by name, and OS/2 returns the selector to the memory segment. This technique is called *named shared memory*.

Semaphores

Semaphores allow multiple threads to control access to serially reusable resources in an efficient, controlled manner or to signal the occurrence of an application-defined event between two threads. For example, access to the logical video buffer can be controlled by semaphores. Using the API calls, an application can control which process can update the screen. This eliminates the potential problem of two processes updating the screen simultaneously.

Figure 2-5 on page 2-7 illustrates two processes communicating using a semaphore.

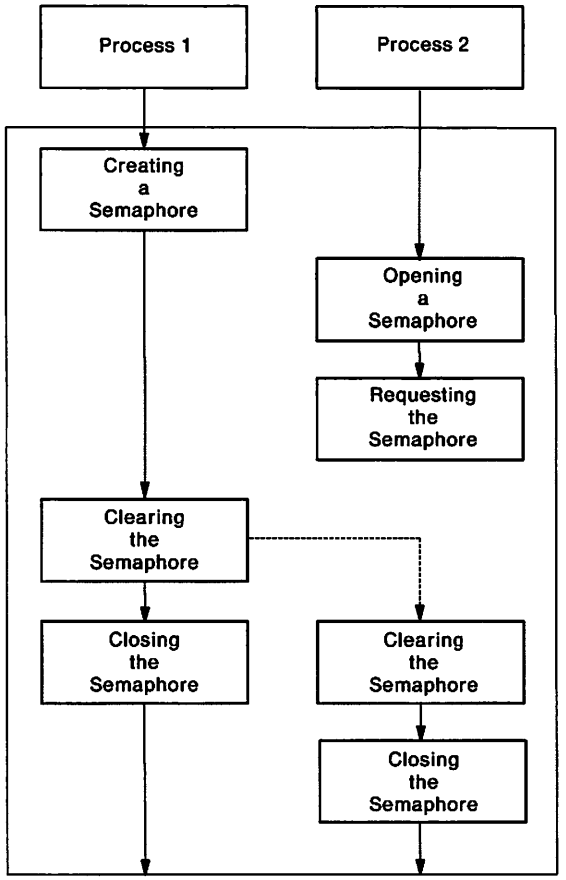


Figure 2-5. Communicating using a Semaphore

There are two types of semaphores, the *system semaphore* and the *random access memory (RAM) semaphore*. There also is a sub-class called a *fast-safe RAM semaphore*.

A system semaphore is a full-function mechanism managed by OS/2 that can be used by multiple processes and threads that do not share memory. A system semaphore is defined using the file-naming conventions and resides in a buffer in memory. A system semaphore has two states, owned and unowned. Using the semaphore API calls, a thread can claim ownership of a semaphore by requesting access to and waiting for the semaphore to clear. However, another thread cannot claim ownership of a semaphore until the owning thread releases it. If a semaphore is defined for global access, any thread knowing the semaphore's name can modify its state.

When a process ends either normally or abnormally, OS/2 manages the orderly freeing of resources. The system notifies threads waiting for access to a system semaphore that the resources are available. System semaphores require more overhead than RAM semaphores.

A RAM semaphore is a minimum-function, high-performance mechanism that allows two or more threads to communicate using a double-word in physical memory. RAM semaphores are suitable for signaling events. RAM semaphores do not need to be created, opened, or closed, so they require less system overhead than system semaphores. Because the system does not manage RAM semaphores, they are best used among threads in a single process, but this is not a requirement. If the thread owning a RAM semaphore terminates without freeing the semaphore, OS/2 does not notify threads in other processes that the semaphore is free.

The fast-safe RAM semaphore enables a semaphore to function with the performance characteristics of a RAM semaphore and the safety characteristics of a system semaphore. To ensure that all individual threads' claims to resources are relinquished when a process ends, an exit list routine should be written before a request for a semaphore is issued. The exit list routine overrides individual threads' claims of ownership and frees the resources for other processes to use.

Managing File I/O

The file system in OS/2 allows the application to organize and maintain data on external devices and provides a logical view of the storage media. The file system relieves the application of being familiar with the characteristics of each device. OS/2 now supports multiple, compatible file systems. These include the new installable High Performance File System (HPFS) and the traditional file allocation table (FAT) file system. HPFS manages large disk media in a fast and consistent manner and supports files with new long file name characteristics.

The file system is considered a subsystem. See "Managing Subsystem I/O: Video, Keyboard, and Mouse" on page 2-8 for more information about subsystems.

The file system is arranged in a hierarchy on the physical disk. The disk also can be subdivided into two or more logical disks, or partitions, each with its own hierarchy.

The basic element in the file system is the *file* itself. A file represents a serial stream of charac-

ters. A *directory* is a hierarchical collection of files, with at least one *root directory* at the top of the hierarchy on each logical disk. Directories can contain other directories, called *sub-directories*, which are collections of files at a lower hierarchical level than the root directory.

An application can manipulate the contents of a disk to organize it into logical groups of files. OS/2 provides API calls to allow the application to create, open, read, write, and close files and to create and delete sub-directories. Once a file is opened, it is assigned an identifier called a *file handle*, which can be used by other processes to refer to the file.

OS/2 also supports the extended attributes mechanism that allows an application to attach various information to a file object. Extended attributes consist of two parts: an ASCIIZ text name, and an arbitrary value. For a standard set of conventions for naming extended attributes and defining data types see the *Programming Guide*.

Because any application in the multitasking environment can issue API calls, it is possible that two applications can request access to the same file at the same time. To prevent such conflicts, OS/2 provides facilities that, upon request, control access to the information stored in files. The process that opens a file can define how other processes must share the file. If the file is large and needs to be used frequently, a procedure called *file locking* permits a process to protect only small portions of the file, leaving the rest of the file available.

Managing I/O for Handle-Based Devices

OS/2 must provide support for external devices other than the disk. OS/2 supports such devices independently with a model similar to that used for files. Like a file, a standard device represents a stream of characters. In most cases, devices look like files to the application. Like files, devices are identified with ASCII names. The *Programming Guide* describes specific naming conventions.

OS/2 provides file subsystem API calls to allow the application to create, open, read, write, and close devices. I/O performed on character devices must be processed serially. When a device is opened, it is assigned an identifier called a *device handle*. The device handle can be used by a process for input and output and other read and write operations to and from the device.

Each process started in the OS/2 environment is provided with a standard set of device handles. These device handles identify the system's standard input device (usually the keyboard), the system's standard output device (usually the screen), and the standard error device (usually the screen) to which error information should be written.

File handles, device handles and pipe handles are indistinguishable to an application. The advantage of using the standard device handles is that data stored in files can be redirected to devices without intervention by the application. This also means that one application can redirect its output data stream to another application's input data stream using a pipe.

However, in the Presentation Manager environment, the user can generate multiple random events with the keyboard and mouse. The meaning of a keystroke or character is determined by the relative location of the mouse pointer on the screen. Such input cannot be redirected. Because the user requires a highly responsive interface, it is not practical to rewrite the screen every time the user moves from one area to another.

See "Processing Messages" on page 3-4 for information on handling input in the Presentation Manager environment.

Managing Subsystem I/O: Video, Keyboard, and Mouse

Like the file subsystem, each of the other three subsystems, the video, keyboard, and mouse, provides a set of API calls through which the application can request services. These API calls begin with the prefixes Vio, Kbd, and Mou, respectively.

All input and output from the application and from the system itself is resolved at the subsystem level. The subsystem generates calls to the OS/2 device driver. See "Device Drivers" on page 2-9 for more information about OS/2 device drivers and "Presentation Drivers" on page 3-12 for information about Presentation Manager drivers.

The idea of sessions must be examined in the context of subsystems. Remember that a session represents a logically separate unit of screen, keyboard, and mouse and their related processes. In the OS/2 environment, the session manager is responsible for coordinating the subsystems so that I/O flows to the correct application. See "Processing Messages" on page 3-4 for information about the Presentation Manager environment.

OS/2 maintains a logical video buffer, which is a shadow of the screen contents, for each session. The keyboard subsystem maintains an input queue for each session. The mouse subsystem maintains an input event queue for each session. The session manager moves the input focus from one queue to another as the user switches sessions.

The user sees a group of multiple virtual consoles. The application sees a dedicated keyboard, video, and mouse. The subsystem sees a logical video buffer, a keyboard input queue, and a mouse event queue.

The API calls to the video subsystem can be issued at any time, even when a session is running in the background. The video interface supports both text and graphics modes. In general, the Vio interfaces support the following activities:

- Manipulating character strings
- Accessing the logical display buffer
- Accessing and sharing the physical display buffer
- Synchronizing screen updates
- Placing messages on the screen from a background session
- Controlling the video mode.

The video subsystem interfaces have been extended to support the needs of applications using the windowing facilities of the Presentation Manager. Those services are highlighted in "Handling Alphanumeric Output" on page 3-10. They are described fully in the *Programming Guide*. The *Programming Guide* also provides guidance for using the correct set of Vio interfaces for each of the four types of supported application.

The keyboard subsystem can receive character string input from the keyboard. This input function is the corollary of the character string output function supported by the video subsystem. The Kbd service is similar to that offered by the file system services, but the Kbd service does not allow the input to be redirected. However, it does allow the subsystem to intercept and manage the session's keyboard input buffer (KIB) contents.

The mouse subsystem supports a range of services for managing the pointing device on the screen. The Mou interfaces include:

- Opening and setting the state of the logical mouse
- Determining which events should be sent to the input queue
- Querying mouse data
- Querying the state of the input buffer
- Reading data from the input buffer
- Manipulating the pointer on the screen.

Replacing a Subsystem

The services of each subsystem are implemented by way of a dynamic link run-time library and a device driver. Therefore, a system extension, which can be part of the application or a different process, can register itself to handle application requests made by way of the API calls. See "Dynamic Linking" on page 2-10 for information about dynamic link libraries.

Device Drivers

Device drivers are the software interface between the subsystems and the hardware. It is really the device drivers that allow the application to maintain device independence. If the device driver layer did not exist, the operating system would have to be rewritten every time a new device is added.

There are two types of devices, *character* and *block*. A character device is data-stream oriented, and I/O is performed in a serial manner. A block device normally holds large amounts of data that can be accessed either serially or randomly. Block devices include, among others, disks and virtual disks. From the application's viewpoint, transfers of blocks of data are transparent. The application performs I/O using the file system API calls. The file system maps the serial I/O requests to random block device requests.

In the multitasking environment, multiple applications share resources, including devices. The device driver guarantees orderly access to a device by managing the device on behalf of all applications sending data to or receiving data from

the device. The application uses an API call to communicate with an API subsystem. The API subsystem uses I/O control (IOCtl) to communicate with the device driver.

The device driver often needs access to the system services, such as memory, inter-process communication mechanisms, or the processor. A set of services, called device helper routines (DevHlps) provide such services. The DevHlps provide a multitude of services to the device driver, including:

- Managing the device queue
- Synchronizing threads
- Managing memory
- Managing processes
- Managing interrupts
- Handling the timer
- Monitoring data buffers.

In addition to the DevHlps, a series of services beginning with the prefix DosMon are available to create an application that monitors the data streams passing through a character device driver. The monitor applications can examine, add, or delete elements in the data stream.

The 80286 hardware protects each application from other applications using a protection, or *privilege* model, managed by OS/2. The premise of the privilege model is that the executing application can access data only at its own privilege level. The application cannot access the operating system's data, but the operating system can access the application's data.

There are four protection, or privilege levels, 0-3, with 0 being the most privileged. OS/2 and the system device drivers run at privilege level 0 while applications run at privilege level 3. Privilege level 1 is reserved. To maintain the integrity of the system, applications generally access a device by means of a device driver.

If a device driver does not provide adequate I/O service, OS/2 provides a mechanism by which the application or subsystem can access a device directly. This mechanism is called an I/O privilege level (IOPL) code segment. Using IOPL, the application can perform I/O to and from a hardware port.

This discussion only highlights the information on device support provided in *I/O Subsystems and Device Support Volume 1: Device Drivers*. In addition to containing a complete description of I/O services available using the OS/2 system device drivers, *I/O Subsystems and Device Support Volume 2: Presentation Driver Interface* also describes additional printer, plotter, and video subsystem support provided by the *presentation drivers*, the set of interfaces available to support applications running in the Presentation Manager environment.

Dynamic Linking

In many systems, such as DOS, inter-segment references must be resolved within the application's executable file. This means that the linker resolves external references and sends a single executable file containing all called subroutines to the loader. The loader brings the executable file into memory and fixes it up to ensure the inter-segment references are valid. In the OS/2 environment, resolution of external references can be delayed until load time or run time.

Dynamic Linking at Load Time

In the OS/2 environment, an application can reference segments that are not part of its executable file. The LINK utility, available with the OS/2 product, reserves placeholders in the executable file for information about external segments. Until load time, the code, data, and stack segments that comprise the application's executable file reside on disk. When the loader receives a request to execute the application, it must build an LDT and allocate memory for the application to run. It also fills in the addresses and other information about the application's external segments.

In "The Application Programming Interface" on page 2-2, we discuss how OS/2 implements the API calls using dynamic linking. From the application's view, the code for each API call is a subroutine that can be called by name using an inter-segment far call. The subroutine that implements the API call is not included as part of the application's executable file. Instead, statements providing information about external subroutines to the linker are contained in an ASCII text file called a *module definition file*. The relocation pointers to the subroutines are contained in a *dynamic link library*. The loader fills in the addresses of the subroutines and fixes up the executable file so that it is ready to execute.

A dynamic link library allows many applications to share routines. This permits each application's executable file to be smaller and allows applications to be updated without re-linking subroutines. If swapping is enabled, sharing common routines in a dynamic link library saves the overhead incurred by moving identical segments in and out of memory.

Common routines that should be accessible to other applications can be placed in a dynamic link library. A dynamic link library routine has the same format and structure as an executable file. An application is written as though the subroutine is in a different code segment and references to the subroutine are made with a standard inter-segment (far) call. At compile time, the compiler (or assembler) generates an external reference point in the object files. Along with the dynamic link library object files and optional library files, a module definition file is mandatory input to the linker. The module definition file tells the linker that the library it is creating is a dynamic link library and defines the subroutines that are available for *export*. The process of building a dynamic link library is described in detail in the *Building Programs* book.

Another application can access the dynamic link library by linking with a module definition file containing the subroutines it needs to *import*. Or, you can create an *import library*.

An import library resolves the application's external references by supplying pointers to the subroutines in a dynamic link library. We recommend using an import library rather than declaring each imported subroutine in a module definition file. If the system changes or subroutines are moved from one dynamic link library to another, the module definition file must be rewritten. An import library generates the addresses automatically. Created using the IMPLIB utility in the tools package, IMPLIB takes the module definition file that you used to create the dynamic link library and generates the import library file as output.

Loading on Demand

To complement the dynamic linking services, segments in a dynamic link library can be designated to be loaded only if the application explicitly requests them at run time. Since the format and structure of a dynamic link library are the same as an executable file's, individual segments can be defined in the module definition file as loadable *on demand*. This means that the segments are not brought into memory unless the application explicitly requests that they run. This facility is particularly suitable to error routines.

Dynamic Linking at Run Time

The disadvantage of load-time dynamic linking is that all references must be known when the application is written. The names of external subroutines are hard-coded in the application's executable file or in a module definition file. Dynamic linking at run time allows the application to call segments dynamically, allowing the application to perform some processing before requesting a dynamic link routine it wants to use. The API calls allow the application to locate a dynamic link routine in the file system, determine if it has been loaded, obtain a handle to it, request addressability, and free it.

Managing Memory

Because OS/2 manages memory, it can discard any application's data, code, and stack segments in physical memory to disk, depending on how often they are used and the demands placed on physical memory by other applications running in the system. In addition, OS/2 alleviates the potential for external fragmentation by compacting segments to fill holes left by allocation and deallocation requests. See "The OS/2 Operating Environment" on page 2-1 for more information.

However, using API calls, an application can dynamically allocate and deallocate memory as it runs on the system. The application can determine how much memory is available, allocate a memory segment up to 64KB in size, deallocate a segment, and change the size of a segment. In addition, the application can allocate and deallocate multiple memory segments of 64KB at a time to accommodate large data structures, although this activity should be used with discretion because it can impair system performance.

In many cases, processes request and free small portions of memory very frequently. Because frequently allocating and deallocating segments consumes overhead, the application can initialize an allocated segment as available for suballocation. The system then keeps track of which portions of the segment are in use and which are not. When a process makes a request for memory, OS/2

returns the index within the segment where memory space is available. Suballocated pieces of memory are not subject to system swapping or compaction.

Although the system frees all memory when a process terminates normally or abnormally, it is good programming practice to free memory segments when a process ends normally.

Programming Considerations for the 80386 Microprocessor

When executing on a system with an 80386 Intel™ microprocessor, OS/2 preserves the 80386 extended registers across all context switches. These context switches include thread switches, process switches, and preemption due to a hardware interrupt. The registers preserved include:

- EAX
- EBX
- ECX
- ESI
- EDI
- EBP
- 32 bit flags register.

These additional 16 bit 80386 segment registers are also preserved:

- FS
- GS.

Note that this does not imply that segments of greater than 64KB are supported. It does imply that the high order word of the following registers will be destroyed on a context switch.

- ESP
- EIP.

Text Messages

OS/2 provides a convenient facility for keeping text message files separate from the application's executable file. Keeping the message file separate allows the messages to be translated easily to other national languages. Message files can be loaded only as needed and can serve more than one application.

Note that the messages referred to here are *not* the same messages as the ones used in the Presentation Manager context. See "Message Box" on page 3-6 for the alternate method of displaying text messages in the Presentation Manager environment.

These text messages display error, help, prompt or general information to the OS/2 application user. The text of the message is contained in an ASCII file. The MKMSGF utility in the tools package converts the text file to a binary file. A set of API calls enables the application to retrieve the message from the binary file created by the MKMSGF utility, insert the string of text held in a buffer into the body of the message file, and display the message to the user.

The MSGBIND utility in the tools package can be used to bind the message file as a data segment to the application's executable file, thus ensuring that it is accessible for immediate display.

System Limits

Certain operating system resources such as memory segments, file and device handles, directory handles, processes, threads, sessions, active timers, system semaphores, and queues have limits. You should be aware of these limits if you program large applications or applications that work with other applications.

Generally, the limits are high enough so that no single application will reach them. However, system resources are limited by the amount of disk space and system memory a system contains and the version of OS/2 being used.

An appendix in the *Programming Guide* lists the system limits for several operating system resources.

Trademark of Intel Corporation

Chapter 3. Presentation Manager Concepts

This chapter describes the Presentation Manager, the presentation interface that implements Systems Application Architecture (SAA) in the OS/2 environment of the PS/2 family. The Presentation Manager implements the Common Programming Interface (CPI) component of SAA, which facilitates program portability to other environments supported by SAA, including VM and MVS on the System/370™ and Operating System/400™ on the Application System/400™.

This chapter describes the concepts and facilities that help you plan and design a Presentation Manager application. They include:

- Systems Application Architecture
- The windowing environment
- Structuring the application
- Handling mouse and keyboard input
- Processing messages
- Providing the user interface
- Developing the help interface
- Handling application resources
- Managing graphic objects
- Handling graphic output
- Handling alphanumeric output
- Producing hard-copy output
- Exchanging data among applications
- Presentation drivers.

Systems Application Architecture

SAA is based on the premise that applications with a common appearance and consistent behavior will enable users to be productive and efficient. The benefit to you is that consistency in application interfaces and structure make programming skills learned in one operating environment transferable to other supported environments.

SAA guidelines are best applied to applications that are widely used, address common needs, and perform common functions, such as:

- Payroll
- Accounting
- Word processing
- Calendar
- Decision support
- Scientific
- Engineering
- Graphics
- Message switching.

The *Systems Application Architecture: Common User Access: Advanced Interface Design Guide* book in the SAA library describes the components of a common user interface, standard window types, actions, action bar choices, and pull-down menus. If you want to know more about creating an application that is similar in appearance to other applications, read the books in the Systems Application Architecture library identified in "OS/2 and Related Product Libraries" on page vi.

The CPI *Common Programming Interface: Presentation Reference* lists the common programming interfaces, including the operating environments in which each interface is supported. Over time, the number of supported environments is expected to increase. The *Common Programming Interface: Presentation Reference* also defines general rules for developing an application that conforms to SAA guidelines, including structuring the application, displaying the application's data on the screen, and handling messages, input, output, graphics, and picture interchange.

If you design and write your application following the programming techniques recommended in the *Programming Guide*, the application will be similar in appearance to others written in compliance with the SAA guidelines. Using the API calls does not enforce compliance with the guidelines, but many API calls do implement the CPI component of SAA.

Although many of the system's defaults, such as the standard control windows, enable compliance, you must know the SAA guidelines and understand how to extend and adapt them to meet your application's needs.

Trademark of IBM Corporation

© Copyright IBM Corp. 1989

The Windowing Environment

A large set of API calls is available for controlling *windows*, an area of the screen through which interaction is presented to the user. Function calls that begin with the prefix Win allow an application to create, size, move, and control windows and their contents. The *Programming Guide* describes common programming techniques for managing the windowing environment, sometimes using detailed walk-throughs of the sample applications in the tools package.

Creating and Classifying Windows

A window and its associated *window procedure* are considered a program object. A window procedure "represents" a window in the sense that the window procedure controls all aspects of the window, such as what it looks like, how it responds to changes, and how it processes input. See "Processing Messages" on page 3-4 for more information about the window procedure.

A *window class* is a set of windows that has the same window procedure to implement them. Many windows can belong to a window class. The windows can differ from one another only in the data they process. If multiple applications have need for the same type of window, implementing common window classes is an efficient means of using system resources.

OS/2 provides many pre-registered window classes, which conform to guidelines defined in the Common User Access. These include window classes that implement windows such as the standard window described in "Standard Window" on page 3-5, menus, scroll bars, and dialog boxes. If a pre-registered window class is not provided, the application must register the class at the process level. Several API calls are available for applications to reserve small amounts of memory for windows in classes registered by the application. This area, called the *window words* area, typically is not very large. If a window expects to handle large amounts of data, the data should be held in a segment of memory and referenced from the window words area.

A window class can be defined as public or private. However, note that windows created in either class can be used by any process in the system. Public and private refer only to the window class at the time the window is created. Only the process with which a private class is registered can create a window for that class. The class name must be unique to the process. However, other processes can register private classes with the same class name. Any process can create a window in a public class. Window procedures for windows in public classes must be available to all processes. Thus, such window classes should be defined in dynamic link libraries. Public class names must be unique for each process.

All windows have certain attributes. Each window is identified by a *window handle*. Each window represents a *rectangle* describing the size and position of the window on the screen. The size of a window is defined in pels relative to the origin of the parent window. The origin of a window, the lower, left corner, is the 0,0 coordinate in a set of x− and y−axes. The x− and y−coordinates define the top, bottom, and sides of the window. The coordinates range from −32768 to +32767 pels in each direction, so the maximum size that can be specified in any direction is 65535 pels.

Similarly, the application can position a window by defining its relative distance from the point of origin (0,0) of its parent window. If the application positions a child window outside its parent window, which is permissible, only the part of the child window within the parent window will be visible on the screen.

Using a set of API calls, the application can modify the behavior of a window in a window class or create a new window class from an existing one. This allows the application to modify the behavior of a single window without rewriting its complete window procedure.

Defining Window Relationships

Windows are a means of sharing, subdividing, and organizing the screen. The common analogy is that multiple windows on the screen are like many pieces of paper on a desktop. In the analogy, the desktop is the area which comprises the background of the screen. Windows, like papers, can be arranged to lie on top of one another and overlap. If they overlap, the bottom papers can be either partially or completely obscured. The application can define windows in the window hierarchy using the API.

Figure 3-1 on page 3-3 illustrates the window hierarchy as it appears on the screen.

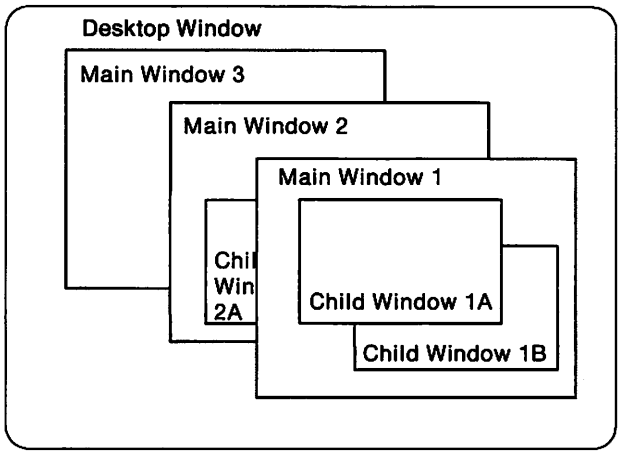


Figure 3-1. Windows on the Screen

The *desktop window* is at the top of the hierarchy. Below the desktop window are the top-level windows, called *main windows*. Main windows can overlap one another and, at times, a main window can be obscured completely. Operations on one main window do not affect those on other main windows.

Figure 3-2 illustrates the hierarchical arrangement of windows created by the application.

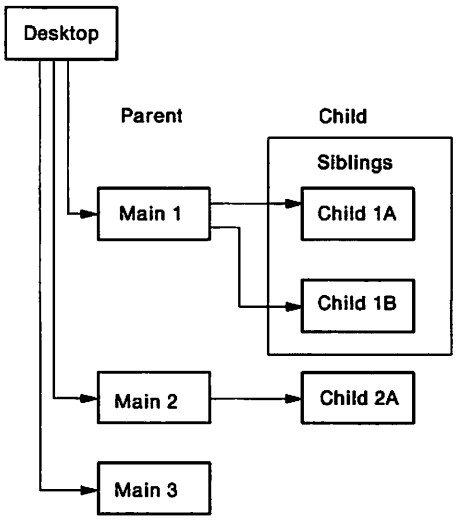


Figure 3-2. Window Hierarchy

Main windows can create subordinate windows in a parent-child order of descendancy. A child window is completely contained within its parent window. A child window is always *clipped* to its parent window, meaning that only the part of a child window that lies within the parent window is visible.

Windows that share the same parent window are called *sibling windows*. Like main windows, sibling windows overlap one another. Linked together in a list, the first window in the list is considered the top-level window. This order represents the *z−order*, or the z−axis.

The application can define another relationship in addition to the hierarchical one. When a window is created, an *owner window* optionally can be defined. The two windows must be created by the same thread. The owner relationship varies at different levels of the hierarchy. A child window can send messages to its owner window. If one main window owns other main windows, and the owner window is hidden, minimized, or destroyed, all the owned main windows are hidden, minimized, or destroyed.

A window can be visible on the screen or hidden. When a window is hidden, its size, position, and hierarchical and owner relationships remain the same. However, any drawing in the window must be updated when the window becomes visible again. A window also can be disabled, meaning that it is still visible but unable to respond to (mouse) input.

Structuring the Application

The CPI defines a modularized application structure that includes a main procedure and window procedures. See the CPI *Common Programming Interface: Presentation Reference* for more comprehensive information.

The *Programming Guide* describes in detail, using sample applications from the tools package, the structure common to all applications that use the services of the Presentation Manager. The structure of a Presentation Manager application adheres to the rules defined in the CPI.

Handling Mouse and Keyboard Input

The session manager component of OS/2 manages OS/2 applications running in the Presentation Manager environment, including their input and

output operations. However, the Presentation Manager component of OS/2 handles Presentation Manager applications, including their input and output operations. The Presentation Manager handles all input as *messages*, which are packets of data.

The Presentation Manager supports user input from the keyboard and *mouse pointer*. The mouse pointer is the screen pointer object associated with the mouse pointing device. Mouse input, as a result of pressing a button, usually is directed at the window under the mouse pointer. The precise position is called the *hot spot*. The hot spot is a single pel located within the mouse pointer. However, the mouse pointer also can be moved across the screen, and OS/2 provides support for that input. The application can direct all mouse input to a single window called a *mouse capture window*. A mouse capture window allows the application to track all input from the mouse pointer no matter where the mouse pointer is moved on the screen.

The appearance of the mouse pointer is defined in a *bit map* defined in "Handling Graphic Output" on page 3-9. The application can create, display, move, and destroy the system pointer.

Keyboard input is sent when any key on the keyboard is pressed. All keyboard input is directed to one window at a time. The window receiving keyboard input is called the *active window*. A main window or one of its child windows is responsible for keeping the window with the focus visible on the screen.

The *cursor* is a symbol displayed within a window that indicates the location where characters generated from the keyboard will be placed. The cursor can be moved to any location within a window. The size and position of the cursor are defined in window coordinates relative to the window in which the cursor is located. The cursor is clipped to the window. The application can create, display, move, and destroy the system cursor.

Processing Messages

The CPI defines an application structure that uses a system of queues and application window procedures for processing messages, which are packets of data. Do not confuse these messages with those discussed in "Message Box" on page 3-6.

The message system supported by the Presentation Manager conforms to the CPI and is fundamental to the smooth operation of the Presentation Manager environment.

Figure 3-3 illustrates the Presentation Manager message processing system.

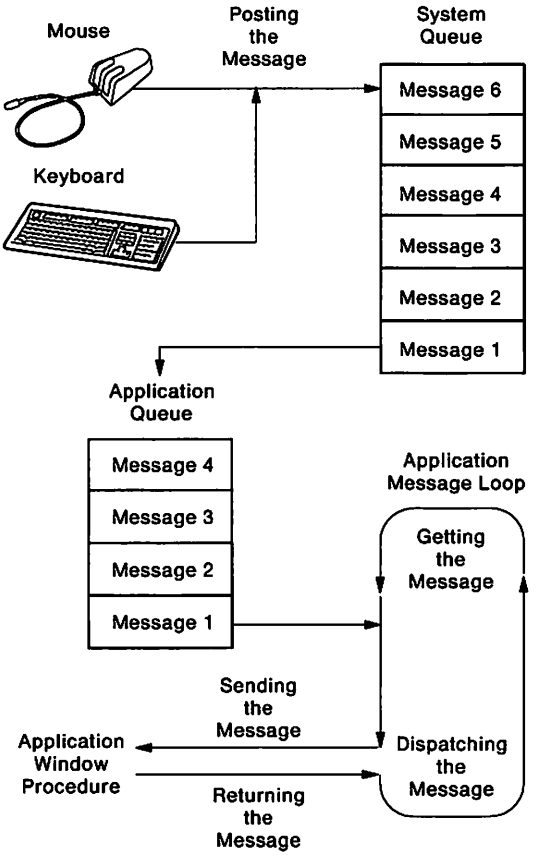


Figure 3-3. Processing Messages

In the Presentation Manager environment, each input event originating from the mouse or keyboard is delivered to an application as a message. A message cannot be processed until all previous input has been processed. This is because the specific application and window for which input is intended are not known until all preceding input has been processed.

All input is placed first in a single queue called the *system queue*. The system queue, which is shared by all applications in the system, receives messages generated by the user from the mouse and keyboard. The system queue can hold approximately 60 key presses and mouse clicks. The system queue is capable of receiving events from many sources, including the system itself, the

timer, and other applications. However, only user input is processed synchronously.

The system queue buffers user input so that no input is lost if the user enters data faster than the application can process it. Generally, input is processed in the order in which it appears on the queue, but the application can change the order by *filtering* the input. Filtering is performed using API calls, but should be performed with discretion because the processing of one event often changes the context for the next event. Note that in the Presentation Manager environment, this facility replaces the mouse and keyboard device monitor.

Each thread that receives input has an *application queue* allocated by an API call. The application queue does not receive user input directly, but can receive other messages directly, such as messages from the system or timer. Messages are removed from the application queue when the thread for which it is destined "gets" it. The messages are prioritized if more than one is waiting. If there are none, the thread is suspended until a message arrives.

Note that if you have structured your application so that one thread remains responsive to user input while other threads continue processing other work, the system is used most efficiently. To be considered responsive to the user, input processing should take no longer than 0.5 seconds. That is, the thread handling input should check for the next message on the queue within that time.

The window procedure code is activated as a response to a message. A window procedure receives messages as input parameters. A window procedure can control more than one window. The first parameter identifies, by handle, the window for which the message is intended. The second input parameter indicates which type of message has been sent. The last two parameters contain data associated with the message. The window procedure deals with the message, and a return value is sent back to the sending code. A window procedure must respond to all messages sent to it, even if the response is to send the message back to the system's default window procedure.

There are many types of messages, each with a unique identifier, and applications can define their own message types using a range of identifier values.

Providing the User Interface

The Common User Access is a set of guidelines for designing and writing the application's user interface. The guidelines cover standard action bar items, interaction techniques, and window types. You should use *Systems Application Architecture: Common User Access: Advanced Interface Design Guide* as your guide in deciding how to design your application's user interface.

Some of the services of the Presentation Manager, if allowed to default, help to enable a consistent user interface among applications designed and written following the guidelines defined in the Common User Access. Consistency also is enabled by selecting appropriate options. Ensuring consistency is the responsibility of the application developer.

The Common User Access defines two types of windows: *primary* and *supplemental*. There are two type of supplemental windows: *secondary windows* and *dialog boxes*. The primary window is the main interface point between the application and the user. A secondary window is generated from a primary window. The application user can carry on a dialog in the secondary window in parallel to the dialog in the primary window. For example, in a text editor, a secondary window might contain an option that lets the user change document format while the primary window contains the main editing activity. Secondary windows also are used to present help information that relates to dialogs in primary windows. Users can switch back and forth between primary and secondary windows.

A dialog box window extends a dialog between a user and a primary or secondary window.

Standard Window

The Presentation Manager supports a *standard window* whose elements generally conform to guidelines in the Common User Access.

The standard window, which the application controls using the API calls, can include all or part of the following elements:

- Sizing borders
- System menu icon

- Title bar
- Maximize/minimize/restore icons
- Action bar and pull-downs
- Scroll bars
- Client area.

Figure 3-4 on page 3-7 illustrates a standard window and its elements.

The standard window is created using a standard *frame window*. The elements that comprise the standard window, such as the title bar and the action bar, are child windows of the standard frame window. The child windows are called *control windows*. The system maintains a set of pre-registered control windows that can be used to perform I/O by any application in the system.

Control windows are no different from other windows in the system, and the application can manage them using the window management functions. Each control window is identified by its own window identifier, and the application can query the system to determine the parentage of a control window. Each control window has a specific set of messages. Control windows usually are part of a dialog window and are loaded from a dialog template, but the application can create them using the API. The *client area*, and its associated *client window* must be created and registered by the application.

The standard frame window and the control windows are implemented with standard pre-registered window classes. The standard frame window is responsible for managing the control windows and the client window as the user interacts with them. The frame window also is responsible for routing messages to the appropriate control and client windows.

A window is the area of the screen through which information is displayed. In the Presentation Manager, a primary window is a standard window. A secondary window is a control window or the child of a main window.

Dialog Box

The *dialog box* that appears on the screen usually is generated by the user selecting an option from the action bar which generates a pull-down menu. Selecting one of the options in the pull-down menu generates a dialog box. The dialog box can contain any of the following choices or a combination of them: buttons, entry fields, icons and text, list boxes, and title bars. A dialog box and its supporting window allow the application to gather input from the user. A dialog window usually is created temporarily for special purpose input and then destroyed immediately after use.

There are two types of dialogs, *modal* and *modeless*. A modal dialog retains control until the application issues a call to destroy it. Users are not able to activate other windows belonging to the application until they finish interacting with the modal dialog. A modeless dialog allows windows in other applications to be activated after it has been created.

A dialog window can be created using a *dialog template*. A dialog template defines the position, appearance, and window identifier of the dialog window and each of its child windows. A template can be loaded as a resource or created dynamically in memory. The dialog template can be used to create dialog windows of all window classes. The window classes can contain control windows of all window classes. Or, the application can create its own dialog controls by creating and pre-registering its own control window class.

A dialog window is simply another form of window controlled by a window procedure called a *dialog procedure*. The dialog procedure is responsible for responding to all messages sent to the dialog window, whether by sending them to the control windows or returning them to the default dialog procedure. A set of API calls enables the application to create, load, process, and destroy, dialog windows. The dialog procedure can obtain the handle of its child windows, send messages to them, and process messages and text strings itself.

Message Box

A *message box* is a predefined dialog window available to all applications for displaying messages and receiving input from users. A message box contains a specified caption and message and up to four pushbuttons.

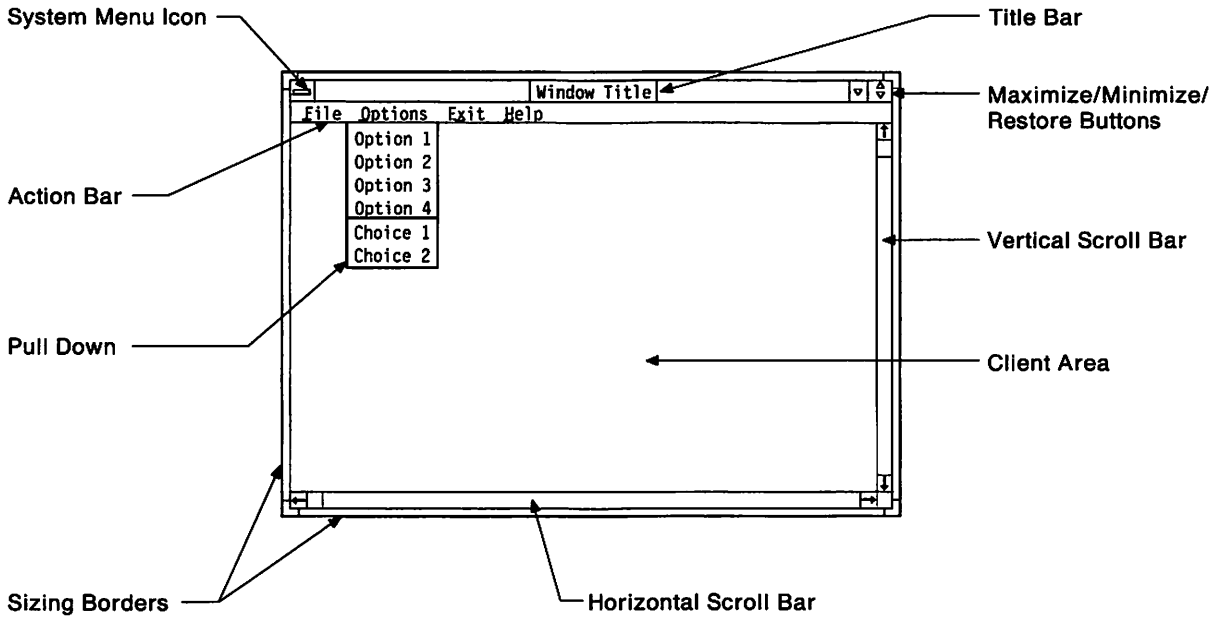


Figure 3-4. Standard Window

Developing a Help Interface

OS/2 now allows the creation of online help information with the Information Presentation Facility (IPF) and the Information Presentation Facility Compiler (IPFC) for Presentation Manager applications. IPF follows CUA guidelines so that the help interface is presented in a consistent format.

Step through the creation of the help interface using the IPF Tag Language. The IPF Tag Language is a *markup* language composed of predefined control words, or tags. Text files, written in the IPF Tag Language, define the help panel and can be created with a text editor. The IPFC reads the markup and translates it into an IPF library format. Since the online help facility provides help information that explains a Presentation Manager application, the use of the help facility is limited to programming languages that can register window procedures. These include C/2, Macro Assembler/2, COBOL/2 and FORTRAN/2 programming languages. The *Programming Guide* provides a complete description for developing an online help interface.

Handling Application Resources

An *application resource* is held in a *resource file*, a text file that provides information that helps to define a window. The resource file defines the names and attributes of the application resources that must be added to the application's executable file. Resources include the following:

Bit maps
A representation in memory of the data displayed on an all-points- addressable (APA) device, usually the screen.

Pointers
The symbol displayed on the screen that can be moved by a pointing device. The pointer is defined in a bit map.

Icons
A pictorial representation of a selection choice. The icon is defined in a bit map.

Fonts
A typeface definition for character sets, marker sets, and pattern sets.

Dialog includes
A definition of a dialog box in a header file.

String tables
A null-terminated ASCII string. A string table is loaded when needed by the executable file.

Accelerator tables
A single keystroke that invokes an application-defined function.

Menu
Supplies the contents of the action bar choices and their associated pull-downs.

Dialog and window templates
The definitions of a dialog box or window containing details of its position, appearance, its window identifier and the identifiers of its child windows.

See the *Programming Guide* for guidance in creating a resource file and *Building Programs* for guidance in building the resource file using the *resource compiler* (RC) utility in the tools package. The RC processes the resource text file to produce a binary file and attaches it to the application's executable file so than an application can access its resources.

Resource Editors

The *Dialog Box Editor* in the tools package allows you to design dialog boxes interactively on the screen and save the definitions in a resource file. The definition of the dialog box is included with other resource definitions in the application's resource file.

The *Font Editor* in the tools package allows you to edit font files interactively on the screen, save the definitions in a font file, and include the font file names in the application's resource file. The font file consists of a header file and a collection of character bit maps representing the individual letters, digits, and punctuation characters that display text on a screen.

The *Icon Editor* in the tools package allows you to create customized icons, pointers, and bit maps interactively on the screen and save the definitions in a resource file. The Icon Editor allows you to work on a large-scale version of the icon, pointer, or bit map while displaying a replica of actual size.

Managing Graphic Objects

Computer-generated graphics is a well-established means of presenting information. The graphics API, which begins with the prefix Gpi, defines the Presentation Manager's extensive capability in support of a graphics presentation.

The *Programming Guide* describes how to create and present graphic pictures. The *Presentation Manager Programming Reference Volume 1* includes the comprehensive, alphabetical list of Gpi calls, their parameters, error returns, and the graphic orders.

Creating and Storing Graphic Pictures

The Presentation Manager provides a set of building blocks that the application can use to construct pictures. These building blocks are called *graphics primitives*. The primitives include lines, markers, images, areas, arcs, and character strings. Using these primitives, the application can construct complete pictures for display or printing. In addition, the application can place the primitives in libraries that can be used as the building blocks in more complex applications. Each primitive can be defined with specific attributes, such as color, type of line, or type of shading.

A *graphic segment* is a means of grouping and storing graphics primitives and their attributes. A graphics segment can contain many unrelated Gpi calls that the application can store and use to rebuild a picture. Although the graphics within a segment usually are related to one another, it is not necessary. For example, a graphic segment can contain the definition of a discrete object, such as the wheel of a car or the window of a house that the application can manipulate as a single graphics object.

A significant advantage of storing graphics in a *segment store* is that they can be re-used. Stored graphic segments can be drawn repeatedly into a single picture and onto many output devices. The application can edit the stored graphics using the API to draw a single segment, part of a *segment chain*, or a complete segment chain. Graphic segments that the application does not store for later use are drawn immediately onto the output device and cannot be reused.

Producing Character Graphics

Using a subset of the API calls that begin with the prefix Vio, the application can produce alphanumeric text. Simple, mono-spaced text produced with the subset of Vio calls is of limited quality and flexibility, but it can be produced quickly. If you

want to mix high-quality text and graphics in your application, use the Gpi calls. See "Handling Alphanumeric Output" on page 3-10.

Assembling Complex Pictures

Assembling a graphics picture requires as many as four stages. At each stage the picture is assembled in a coordinate space. Each coordinate space ranges from −32768 to +32767 units along the x− and y−axes. The four coordinate spaces include:

World coordinate space
Defines a graphics segment or primitive at a specific position

Model space
Conceptually defines the area into which segments and primitives will be brought together to produce a complete picture

Presentation space
Assembles the final picture for display or printing

Device space
Defines the specific characteristics of the device to which the picture is being sent.

Clipping

Clipping is a process that discards the parts of a picture that lie outside a specified *clipping boundary*. Clipping can occur at each stage of the assembly process. The clipping boundaries are application boundaries separate from those defined by the windowing process in which output is clipped to the visible region of the window. If the application does not define clipping boundaries, graphics are clipped to the intersection of the presentation space and the client area of the Presentation Manager window.

Transforming Graphic Objects

Transforming a stored graphic object allows the application to change the object's appearance without changing the definition of the object.

Scaling
Changing the size of an object

Rotating
Changing the orientation of an object

Translating
Changing the position of an object

Shearing
Changing the shear (slant) of an object.

Transforming graphic objects is not mandatory because the system provides defaults, but most relatively complex graphics applications will use the facility. Graphic segments are defined once, but they can be drawn many times, and it is very likely that subsequent drawings will not be required in the same position. To change the position of a graphic segment, the application simply can replace the original set of coordinates with another set of coordinates.

Handling Graphic Output

Graphics Presentation Spaces

The *graphics presentation space* is an area in which graphic pictures are created before being sent to an output device. The graphics presentation space also contains other information about the graphic data it holds. Because most Gpi calls operate on a graphics presentation space, the application first must define a graphics presentation space. Each presentation space is identified with a unique handle. An application can define many presentation spaces.

There are two types of presentation space, a *normal* presentation space and a *micro* presentation space. The normal presentation space accepts all the Gpi calls and therefore is used most of the time. A micro presentation space accepts a subset of the Gpi calls and is used for simple drawings directed once to a single output device. Its advantage is that it saves storage overhead. In addition, the Presentation Manager maintains a limited number of *cached* presentation spaces in storage which can be allocated quickly.

The graphics presentation space contains all the information required to produce a device-independent drawing. Apart from the detail of the drawing itself, the presentation space includes the resources that define the appearance of the drawing.

Device Contexts

Each device to which the drawing is to be sent has unique characteristics. Those characteristics are defined in a *device context*. Each window can have a device context associated with it. The application requests that a device context be associated with a specific window using an API call. To associate a device context with other output devices, the application must issue another API call. Each device context is identified by a unique handle.

A presentation space must be associated with a device context before a drawing can be printed or displayed. The application supplies the handles of the presentation space and the device context with which it is to be associated. A presentation space can be associated with only one device context at a time, and conversely. The application can associate, disassociate, and destroy a presentation space. A device context associated with a presentation space must be closed when the association is over.

The device context can write graphic data to a window on the screen, to a *metafile*, into a *bit map* in memory, or to another device context, which then sends output to a printer or plotter.

Metafiles

A metafile defines the contents of a picture so that it can be used by other applications. Pictures created by an application normally are discarded after the application is run and recreated the next time the application is run. But, because there are times when other applications may want to use the same graphic data, OS/2 provides a retention facility.

Pictures created using the Gpi calls conform to the Mixed Object Document Content Architecture (MODCA) interchange standard. Thus, another application that conforms, even if it is not a Presentation Manager application, can use a picture created for the Presentation Manager environment. The metafile contains all the Gpi instructions that create the final version of the picture. The contents of the metafile are likely to be similar or identical to those of the graphics presentation space.

Bit Maps

A bit map is a representation in memory of the data displayed on an all-points-addressable (APA) device, usually the screen. Bit maps define the contents of a complete physical or logical device. A window, for example, is a logical screen. Because bit maps require no picture generation before displaying their contents on the screen, their greatest advantage is speed. Bit maps are very useful when menus need to be displayed and removed rapidly, the visible region of a picture needs to be restored quickly, cursors or icons need to be generated quickly, areas need to be re-filled, or graphic pictures require animation.

Handling Alphanumeric Output

We said that an application can produce simple, mono-spaced text quickly using a subset of the Vio API calls. To present such text, the application first must create an alphanumeric Vio presentation space and associate it with a device context. An alphanumeric Vio presentation space is a rectangular buffer of alphanumeric characters which can be displayed in a window on the screen. Note that the only device context which can be associated with an alphanumeric Vio presentation space is a screen device context.

API calls enable the application to query the list of supported presentation space formats, the fonts appropriate for each presentation space, and the default fonts for each presentation space. The application must define the size of the alphanumeric Vio presentation space.

Many of the Vio calls from the base OS/2 are available to an application in the Presentation Manager environment. However, some are not compatible. In addition, the Presentation Manager extends the OS/2 Vio calls. See the *Programming Guide* for a list of the exceptions and extensions.

Producing Hard-Copy Output

An application can send alphanumeric and graphic data directly to a hard-copy device (printer or plotter) or it can send the data to the *spooler* for printing when a suitable device is free. The spooler creates print jobs and stores each as a temporary spool file on disk. The advantages of

using the spooler are that it allows hard-copy devices to be shared among a number of processes in the system without interference. In addition, the spooler allows relatively slow printing operations to take place in the background, letting foreground applications continue to interact with the user. See "Presentation Drivers" on page 3-12 for related information about presentation drivers.

There are two main methods for sending data to a hard-copy device. The recommended method is to use the device function calls to create a device context to which the data is sent. This method allows the application to control a print job. For example, it can choose the spool queue to which the data is sent, it can specify a priority for the job, and it can query and subsequently use special features of the output device that is selected by the spooler. In the case of graphics applications, the graphics presentation space in which the data is drawn must be associated with the printer device context.

An application can request *direct* printing (that does not use the spooler) when it creates the printer device context. Although this is normally not recommended, this feature allows specialized applications to use dedicated hard-copy devices that need not be shared with other applications running in the system.

An alternative method is to use the DOS function calls. This method writes data to a spool file if the spooler is running, or direct to the chosen printer if the spooler is not running. This method allows the application no control over the print job and is not suitable for graphics applications.

Exchanging Data Among Applications

Both users and applications running on the system can ask to exchange data with other applications. Data exchange requested by a user is held in an object called a *clipboard*. Requests by an application to exchange data with other applications is called *dynamic data exchange*.

User-Generated Data Exchange

Users can transfer data from one Presentation Manager application to another using the copy, cut, and paste commands. Users first copy selected data into the *clipboard*, or delete (cut) it from the source application, and then insert (paste) it into the target application. The operation also can apply to data that users wish to move from one window to another within a single application. The cut, copy, and paste commands must be supported by a Presentation Manager application as defined in *Systems Application Architecture: Common User Access: Advanced Interface Design Guide*. They are implemented using a set of API calls.

The clipboard is the object that temporarily holds data. Logically, there can be only one item of data in the clipboard at a time, but the data can be in a variety of formats, such as text, metafile, or bit map. The application can define the formats or use one of the pre-registered standard formats. The application can register formats, as it can window classes, so that they can be used by all applications in the system.

The application should support as many formats as possible because it cannot anticipate which format a target application may request. For example, a spreadsheet application should support a spreadsheet format and as many common text formats as possible. However, it would not make sense for a word processor application to support a spreadsheet format because that format is beyond the scope of the operation of a word processor application.

Generating data in all the formats supported by an application can consume a lot of overhead. Generally, data is placed in the clipboard only when a request to paste it is received.

To access and own a clipboard, it must be opened. The thread opening the clipboard has exclusive access until it closes the clipboard. Once data has been sent to a clipboard, it should not be changed. The owning application can change the owner of a clipboard.

The clipboard is owned by the last window that requested ownership. If an owning window is destroyed, data can remain in the clipboard. Before being destroyed, the owning window must

generate its data to satisfy subsequent paste requests.

Application-Generated Data Exchange

The dynamic data exchange protocol allows applications access to one another's data. Dynamic data exchange uses the Presentation Manager message processing system and the shared memory inter-process communication protocol to allow large quantities of data to be passed among applications. API calls to manage shared memory begin with the prefix Dos. See "Shared Memory" on page 2-6 for information on shared memory.

The data is held in a shared memory segment available to designated applications, or all applications in the system. Data is passed in a mutually-agreed-upon format. When the requesting application receives a handle to the data segment in memory, it asks to be advised, by way of a message, if the data changes. If notified of a change, the application can indicate, again by way of a message, if it is interested in the changed data. If so, the application asks that the changed data be sent to it. If not, the application ends the exchange.

Presentation Drivers

Presentation drivers are special-purpose I/O routines which field device-independent I/O requests from the Presentation Manager and its applications. The requests are resolved with the device-specific attributes of the device. In this way, they are similar to OS/2 device drivers. See "Device Drivers" on page 2-9 for information about OS/2 device drivers. In addition, the *I/O Subsystems and Device Support Volume 2: Presentation Driver Interface* book explains presentation drivers in detail.

The presentation driver is a higher-level interface to an output device than the OS/2 device driver. OS/2 device drivers are loaded at privilege 0, the most privileged level. Presentation drivers are loaded at privilege level 2. Both OS/2 device drivers and presentation drivers have I/O privilege.

Presentation drivers are subsystems. They resolve I/O requests from the Presentation Manager and applications in the Presentation Manager environment by interfacing directly with the device or with its device driver. Printer and plotter presentation drivers use the spooler interface beginning with the prefix Spl to pass data and controls to the OS/2 device drivers. Screen presentation drivers pass data and controls directly to the display hardware interface.

Presentation drivers are supplied as dynamic link libraries. When the Presentation Manager is initialized, the screen presentation driver is loaded and enabled automatically. Other presentation drivers, such as those supporting the printer and plotter, are loaded and enabled in response to the application requesting a device context. See the tools package for the printer presentation driver sample program **PMPRT**.

Chapter 4. Dialog Manager Concepts

This chapter describes the Dialog Manager and the Dialog Tag Language. The Dialog Manager supports development of text-based, interactive applications which conform to the SAA Common User Access (CUA). If you are already familiar with ISPF or EZ-VU, you will recognize many of the Dialog Manager functions.

The Dialog Manager provides a common programming interface (CPI) between applications and the Presentation Manager. Dialog Manager functions call Presentation Manager API functions to conduct dialogs on behalf of applications.

Applications developed using Dialog Manager services run in windows supplied by the Presentation Manager. However, Dialog Manager applications may directly call only a subset of Presentation Manager functions. In addition, because of their nature, Dialog Manager applications are structured differently than Presentation Manager applications.

Since the Dialog Manager creates applications that conform to CUA, you can write an application without having to learn CUA rules. However, certain rules require application involvement. Dialog Manager provides services that enable the application to enforce those rules.

The Dialog Manager is based on the SAA Dialog Interface. The Dialog Interface defines elements that are consistent across the SAA environments. Therefore, SAA applications can be migrated to other application platforms, such as the mid-range IBM AS/400. The Dialog Manager contains additional CPIs, or system extensions, that are available only in OS/2. Applications can be designed for cross-system use if these system extensions are not used.

This chapter includes conceptual information about the Dialog Manager, including discussions on:

- Designing the application panel
- Defining dialog elements
- Managing dialog data
- Developing a Dialog Manager application
- Deciding when to use a Dialog Manager application
- Using Presentation Manager functions in Dialog Manager applications
- Providing support for language translation
- Providing Dialog Manager code page support.

This chapter also describes the tag-based language used to create dialog elements.

For a more detailed description of Dialog Manager and Dialog Tag Language, see *Dialog Manager Guide and Reference* and *Dialog Tag Language Guide and Reference*.

Designing the Application Panel

A *dialog* is the interaction between a person and a computer. A dialog is conducted through a *panel* that is displayed in a Presentation Manager window. A panel is defined by the application developer using the Dialog Tag Language. The Dialog Manager automatically displays the panel when requested by the application.

Figure 4-1 illustrates the different panel elements.

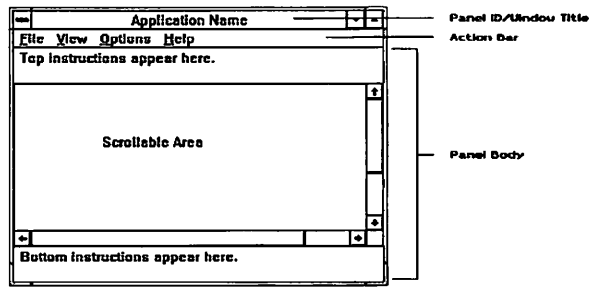


Figure 4-1. Parts of an Application Panel

Elements, such as borders, minimize and maximize icons, system icons, and scroll bars are added automatically by the Dialog Manager. Others are specified by the application developer.

Panel elements include:

Window title
The window title identifies the window. The user clicks on the window title to move the window around the screen.

© Copyright IBM Corp. 1989

Action bar
The action bar is positioned directly below the window title and contains action bar choices. Pull-downs are associated with all action bar choices.

Instruction lines
Top and bottom instruction lines provide directions to the user on how to proceed with the dialog.

Panel areas may be scrollable, and each area can contain many kinds of smaller areas or *fields*.

To conduct a dialog between a user and the computer, the application developer may utilize one or more of the following:

- Input and output fields
- Commands
- Function keys
- Messages and helps
- Variables.

There are several different ways to obtain information from the user. For instance, a data field, where the user enters data, is the most direct way; selection fields, where choices are made, is another way.

Data fields are areas on the panel where the user is asked to provide input. Data entered in a data field may be validated for correct format, for correct number, or correct character.

Selection fields are areas on the panel from which the user can make a selection. An application might also define a group of fields from which the user can choose. Each panel can have one or more of both types of selection fields. The selection can be made by using the keyboard or mouse to select a choice or by typing the initial character of the item to be selected (mnemonics).

Output areas are areas on the panel where a program provides information to the user. The output can be in response to a user request or other action, or a redisplay of information previously entered by the user.

Commands instruct the program what to do next. They can be associated with action-bar pull-down choices or with selection field choices. They can be assigned to keys or typed in a command entry field. The Dialog Manager supports a number of built-in commands. In addition, an application may define private or additional commands and provide alternate support for the built-in commands.

Function keys may be used to initiate specific functions. Applications or users may assign a particular function to a function key. Applications assign functions to function keys in the programs themselves. Users may assign functions to function keys through input fields and save them in a profile to be used whenever the program is run. The function key area on a dialog panel displays the active keys for that panel.

Applications may use messages to identify errors or to provide help. For example, when interactive applications are running, users might require information on how to proceed that is not provided in the body of the current application panel.

Finally, when a user enters information, Dialog Manager uses a variable to pass this information from the panel to the application. Variables are identifiers associated with input areas on dialog panels.

Defining Dialog Elements

A Dialog Manager application consists of one or more programs that use *dialog services* to manipulate *dialog elements*. Programs perform processing requested by the user. They can call dialog services to manipulate dialog elements, such as panels and messages.

Application developers define dialog elements by using the Dialog Tag Language. Dialog elements are described in the following sections.

Panels: A panel definition is a description of the panel. It defines both the content and the characteristics of a panel.

Variables: Dialog services use variables to communicate the various elements of a Dialog Manager application. The Dialog Manager provides a group of services for variable management.

Variables are assigned variable classes. Variable classes specify:

- Values
- Editing operations
- Displayed lengths
- Code pages

- Mappings applied on input or output
- Validation checks.

Messages: Message definitions specify the text of a message to users. There are three types of messages: information, warning, and action. A message can confirm that a user-requested action is in progress or was completed, or it can report an error in the user's input.

Commands: Commands are defined in *command tables*. The Dialog Manager provides many built-in commands. However, an application can be written to override the behavior of the Dialog Manager commands.

Function Keys: Function keys and their meanings are defined in *key mapping lists*. Panel definitions refer to these lists in order to display keys and to perform the specified actions.

The Dialog Tag Language and compiler is used to create *dialog elements*, which make up what is displayed in a dialog window. The Dialog Tag Language is a tag-based language that is used to define panels and other dialog elements. The tags are based on the Standard Generalized Mark-up Language (SGML).

An application developer uses Dialog Tag Language to code, or *mark up* dialog objects in source files that are later formatted by the Dialog Tag Language compiler. For example, the PANEL tag is used to define a panel. The PANEL tag looks like this:

```
<panel name=panel1>Application Name
```

Figure 4-2 shows the relationship between several tags and panel elements.

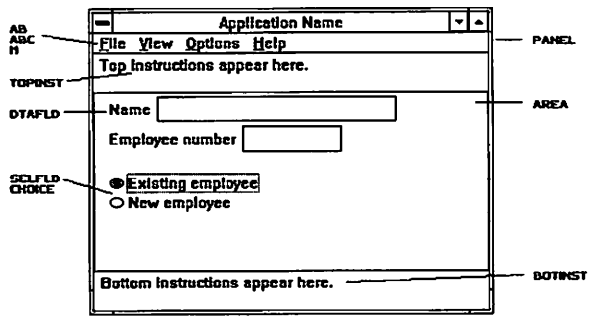


Figure 4-2. Relationship between tags and panel elements

The Dialog Tag Language concentrates on what a panel element represents and how it relates to other panel elements, rather than solely on its appearance. Dialog elements conform to CUA, both in how they are positioned on a screen and how the user interacts with them.

A utility is provided so panels can be displayed without running an application. Application developers can display a panel, verify that the panel elements are correct, and then make changes and recompile their source files.

Manipulating Dialog Services

Applications use dialog services to display panels, manage variables, retrieve and display messages, and control the flow of dialogs. The following dialog services are available to applications. They include:

- Display services
- Control services
- Variable services
- Library services.

Display Services: Dialog Manager display services allow an application to display and remove panels from the screen. Within the context of one display call, many user interactions can occur. The display services automatically handle things such as moving, sizing, and scrolling, as well as variable validation. All of this occurs without any code in the application except the display call.

Control Services: Dialog Manager control services begin and end a Dialog Manager application.

Variable Services: Dialog Manager variable services communicate information between the Dialog Manager and the application. Variable services also allow the application to define, copy, and delete variables.

Library Services: Dialog Manager library services allow an application to specify a library or a list of libraries to be searched for panels. Storing panels in a library simplifies access and retrieval of the panels.

Managing Dialog Data

Associating Data with Dialog Elements

When an application is running, the Dialog Manager uses dialog elements and service calls made by the application to display a panel and then interpret and process user input. When panels are designed and coded, input and output fields are defined using the DTAFLD tag. The DTAFLD tag contains a parameter called DATAVAR that specifies the variable name associated with this field. Figure 4-3 illustrates a panel, the coding for a field, and its associated tag.

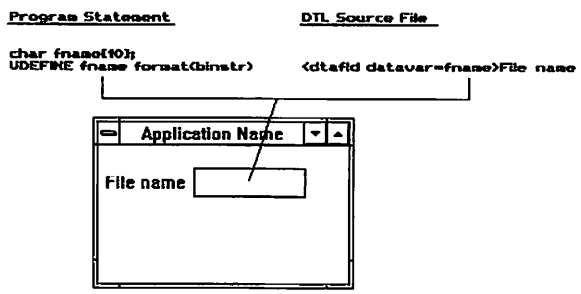


Figure 4-3. Panel with coding for a field and associated tag.

At run time, the application defines this variable to the Dialog Manager using the VDEFINE service. The Dialog Manager associates the variable defined by the application code with the variable defined on the panel and performs any required validations or translations.

Using Variables and Pools

Important Dialog Manager features are dialog variables and dialog variable pools. The dialog variable pool is an area created by the Dialog Manager to manage those variables defined and used by an application. The dialog variable pool serves as the main data communication area between an application and the Dialog Manager. The variables contained in the dialog variable pool are used by a Dialog Manager application to get data to and from panels, place data in messages, and dynamically define command actions.

The Dialog Manager provides services to manage dialog variables and the dialog variable pool.

When a Dialog Manager application calls the service, the Dialog Manager updates the dialog variable pool. The services are:

- VCOPY (copies a variable)
- VDEFINE (defines a variable)
- VDELETE (deletes the definition of a variable)
- VREPLACE (replaces a variable)
- VRESET (resets a pool to empty).

Using System Variables

The Dialog Manager provides a set of system variables that are used to communicate special information between a Dialog Manager application and the Dialog Manager. Some of these variables can be modified by the Dialog Manager application, while others are owned and maintained by the Dialog Manager on behalf of the Dialog Manager application. All system variables begin with the letter Z. An example of a system variable is ZAPPLID, which the Dialog Manager sets to the current application-id.

Translating and Validating User Input

Dialog Manager and Dialog Tag Language provide support for automatically translating and validating user input. Dialog Manager uses *translate* and *assignment lists* to translate input and *check lists* to validate input. Translate, assignment, and check lists are defined with the Dialog Tag Language.

Translate lists convert user input into values that the program can use. For example, to minimize keystrokes the user enters just the first character or two of a name and then have Dialog Manager translate it into a full name.

Assignment lists assign values to one variable based on the content of another variable. It is a way to logically change the value of a variable at the moment the change is required.

Check lists define a list of validity checks the Dialog Manager applies to values the user enters. You can check to make sure values are valid or fall within a numeric range, or belong to the alphabetic character set. Validation takes place when a user exits a field or before values are passed to a program.

Developing a Dialog Manager Application

Develop a Dialog Manager application by first reviewing the application's requirements and CUA guidelines. To conform to CUA, your application must be designed to support an object/action type of approach to information design.

Next, identify the application panels necessary for the application based on what you want the dialog to do. When these panels are defined use the Dialog Tag Language to create the user interface. A text editor is required. Once the tags have been coded, they are compiled using the Dialog Tag Language compiler. Finally, use dialog services and the Dialog Manager run-time code to develop and run your application. Figure 4-4 illustrates the processing of a Dialog Manager application.

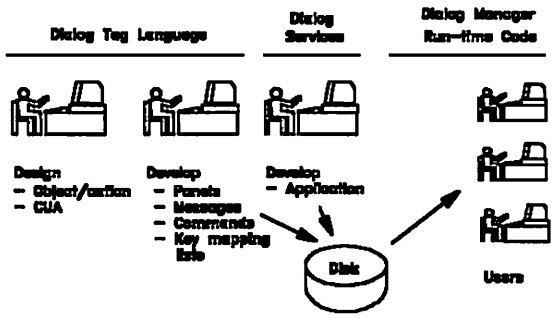


Figure 4-4. Dialog Manager Application Development

The tools package includes a sample Dialog Manager application called DMDEMO. Source code for the application, as well as its dialog elements, is provided. See *Building Programs* for a detailed description of how to build a Dialog Manager application.

Deciding When to Use the Dialog Manager

This section explains some of the differences between the Presentation Manager and the Dialog Manager and gives you information to help you decide which to use for your applications.

Conforming to Customer User Access

Both Presentation Manager and Dialog Manager enable your applications to conform to Customer User Access (CUA). The Dialog Manager provides significant support to implement CUA conventions allowing application developers to focus on CUA concepts. CUA consistency assistance is provided in both the run-time facilities and the Dialog Tag Language compiler. See *Systems Application Architecture: Common User Access: Advanced Interface Design Guide* for details describing how the Dialog Manager helps the application conform to CUA.

Understanding Differences in Programming Environments

There are no major user interface functions provided by Presentation Manager that are not in Dialog Manager. However, certain Presentation Manager functions, such as highly interactive graphics, direct manipulation, and clipboard support, are better suited for a Presentation Manager application.

The Presentation Manager API has a set of services that allows for both application structures and graphics interfaces. This API can be used for most applications.

The Dialog Manager, on the other hand, has a small set of services yet it allows for a Presentation Manager-style of interface. It does this by using the Presentation Manager API calls to present the end user interface and the Presentation Manager application messaging structure in its own design. The Dialog Manager is well suited for the COBOL/2 and FORTRAN/2 application developers, as well as the C/2 developer. Applications can also be written in Pascal and Macro Assembler/2 (on the PS/2).

Other differences in environment and function include:

- Presentation Manager manages all screen input and output. Dialog Manager uses Presentation Manager for all screen input and output.
- Presentation Manager manages the windowing environment. Dialog Manager applications run within Presentation Manager windows.

- Presentation Manager provides a high function and lower level interface. Dialog Manager is a higher level interface than Presentation Manager.
- Presentation Manager has graphical input and output while Dialog Manager has textual input and output.
- A Presentation Manager application must manage its own variables. The Dialog Manager manages variables for the Dialog Manager application.
- The Presentation Manager application must contain code to validate user input. The Dialog Manager provides automatic input validation and translation.

You cannot mix Presentation Manager API calls and Dialog Manager service calls in the same application. For example, you cannot call WinInitialize() or WinCreateMsgQueue() and then call the DMOPEN dialog service. In general, the Dialog Manager application will only need to make Presentation Manager calls inside of user controls and user exits.

Understanding Differences in Application Structure

The structure of a Dialog Manager application differs from the structure as a Presentation Manager application. Presentation Manager applications are event-driven, and use the message queue/polling loop mechanism for application communication. Presentation Manager code is reentrant and written in modular pieces called window procedures. These window procedures respond to messages.

Dialog Manager applications, on the other hand, have a synchronous interface to Dialog Manager. Dialog Manager applications do not know about the message queue because Dialog Manager manages that for the Dialog Manager application. Dialog Manager applications are procedural in nature and use the traditional, procedural style of programming. Dialog Manager applications are a *layer* removed from Presentation Manager and interface with the user through the Dialog Manager.

Creating and Displaying Windows, Panels, and Other Objects

The way that you create resources for an application to display is different in each environment. Presentation Manager uses the dialog box editor to create dialog boxes. Dialog Manager uses the Dialog Tag Language to create panels and other dialog elements.

The Dialog Manager uses many of the graphic capabilities of the Presentation Manager in order to give the Dialog Manager application the same look of a Presentation Manager application. For example, when you start an application, the Dialog Manager uses Presentation Manager to display the primary window. The Dialog Manager then completes the window by displaying a panel in the window.

Dialog Manager panels are composed of several different panel elements. Some elements are added automatically by the Presentation Manager, such as borders, minimize and maximize icons, system icons, and scroll bars. Others are specified by you using the Dialog Tag Language.

Providing Help in Presentation Manager and Dialog Manager Applications

While running an application, users occasionally require additional information about choices, entry fields, messages, or how to proceed with the application. In many cases, you cannot give all of that information on an application panel but must provide associated help panels that the user can access.

Providing help information is similar in both the Presentation Manager and Dialog Manager environments. You use the Dialog Tag Language to create help panels and help indexes for Dialog Manager applications. You use the Information Presentation Facility to create help panels for Presentation Manager applications, as well as Dialog Manager applications. The *Programming Guide* has additional information on the Information Presentation Facility.

Although a Presentation Manager or Dialog Manager application can access the same runtime code that provides help, they access this code differently.

Presentation Manager applications access help at an API level. The Dialog Manager automatically provides access to the help code and performs much of the program logic for you. The interface to the code is handled through the Dialog Tag Language. When you are developing your panels, you include the HELP= attribute within the panel definition. When the user asks for help, the Dialog Manager uses this attribute to access help and to display the desired help panel.

Using Presentation Manager Functions in Dialog Manager Applications

The Dialog Manager provides a set of additional features that you can use when developing your Dialog Manager application. For example, you may suspend your Dialog Manager application to allow for additional processing by using a *forced display exit*. You may add graphics to your Dialog Manager application by defining *user controls*. Define a user exit in order to get control of a Dialog Manager application during panel processing rather than after a DISPLAY service has returned.

To write code that supports a forced display exit, you must understand OS/2 control program functions and how to write multi-threaded applications. To write code that supports user controls and user exits, you must understand Presentation Manager architecture and how to write window procedures. See the *Programming Guide* for more information.

Forced display exit
In certain circumstances, applications execute for long periods of time (greater than one second) without requesting information or interaction from the application user. Examples of this are when some long calculation is required or when the application must retrieve data from a data base or some remote computer. In the OS/2 windowing environment, the end user cannot interact with any other windowed application while a single-thread application runs. In order to allow the user to interact with other applications in the windowing environment, the Dialog Manager provides the forced display exit interface.

User controls
Dialog Manager applications may use the Presentation Manager to create their own panel elements within a Dialog Manager panel. These panel elements are called user controls, and they are defined on a panel using the UC tag. Unlike all of the other panel elements provided by Dialog Manager, the application must write code to support a user control.

User Exits
Dialog Manager applications may have requirements that are beyond the scope of the Dialog Manager. User exits are points of control in a Dialog Manager application that allow the application to gain control during normal panel processing. The application has the ability to modify normal panel processing or override it.

The Dialog Manager provides five types of user exits:

- Action Exit (perform an application-defined action)
- Check Exit (perform application-defined field validation)
- Command Action Exit (execute an application-defined command)
- Translate Exit (perform application-defined data translation)
- Variable Access Exit (use an application-defined variable access method).

Providing Support for Language Translation

National language support provided by the Dialog Manager and Dialog Tag Language gives developers the ability to translate dialog elements into another national language. The translatable dialog elements include panels, messages, commands, and key list definitions. You use the Dialog Tag Language to define dialog elements in a source form that is suitable for editing. After translation, application users can see and use their national language when interacting with Dialog Manager applications.

Dialog Manager run-time support includes proper formatting of date and time, decimal and thousands separator characters, and the like.

Double-Byte Character Set Support

To provide double-byte character set support, the Dialog Manager allows you to specify double-byte character set and mixed-data formats when defining variables to the Dialog Manager. Double-byte character set specifies that the data is in double-byte format. Mixed-data format specifies that the data includes a mixture of single-byte and double-byte characters.

You can also define double-byte character set and mixed-data formats for panel fields and panel text. Double-byte character set support also extends to messages. Panel and message text will be properly formatted.

Providing Dialog Manager Code Page Support

The Dialog Manager recognizes and supports different code pages. The Dialog Tag Language compiler processes source files that were encoded using specific code pages. This code page will either identify a single-byte coded character set or a combined single-byte and double-byte coded character set in a specific encoding scheme.

The tags provide a means of specifying that a given dialog variable contains double-byte characters, single-byte characters, or a mixture of the two.

The Dialog Tag Language compiler detects source file transitions from single-byte to double-byte and vice versa. The objects produced by a Dialog Tag Language compiler are tagged with the identifier of the code page that was used to prepare the tag source file.

Chapter 5. The Five Types of Supported Applications

This chapter defines the five types of applications that run in the five environments supported by OS/2. OS/2 provides operating environments that use both the real and the protected modes of the CPU. In the real mode of the processor, applications are limited to the facilities available to DOS applications. Running in the protected mode, applications can use the facilities supported by that mode of the processor, including large memory addressability, I/O protection, multitasking, and dynamic linking. A Dialog Manager application runs as a Presentation Manager application in the protected mode of the CPU.

This chapter highlights each type of application. The *Programming Guide* describes in detail the sets of API calls available to each of the types of applications. The *Dialog Manager Guide and Reference* describes in detail the set of service calls available to Dialog Manager applications. This chapter also describes the *family application.*

This chapter describes which programming languages are best suited to generating each of the five types of applications.

Presentation Manager Application

Write a Presentation Manager application if you require the following:

- Multiple simultaneous views of data
- Graphic interface
 - Presentation graphics
 - Charting
 - Desktop publishing.
- Multiple fonts
- Ease of use
 - User-driven
 - Intuitive
 - Consistent in appearance and structure
- State-of-the-art technology -- competitive.

The *Presentation Manager* application, if written in compliance with the SAA guidelines, conforms in general appearance and behavior with other applications. This means that the application's graphic user interface looks and operates, with some exceptions, consistently with the Presentation Manager user shell and other applications developed following the guidelines in the Common User Access.

The Presentation Manager application explicitly creates one or more windows in which it presents its functions and data to the user. The Presentation Manager application must be written to share the screen with other applications and must be structured in compliance with the guidelines described in the *Programming Guide*. To stay responsive to the user, it must use the message processing system supported by the Presentation Manager component of OS/2. The Presentation Manager application uses resources such as menus, accelerator tables, string tables, dialog boxes, icons, pointers, bit maps, and fonts defined in resource files to help define windows. Data in a Presentation Manager application can be shared with other Presentation Manager applications and some non-Presentation Manager applications.

The Presentation Manager application runs in the protected mode of the processor and can use the full set of Presentation Manager services, including windowing, graphics, large memory addressability, I/O protection, multitasking, and dynamic linking.

A Presentation Manager application can be written in C/2 Version 1.1, Macro Assembler/2 Version 1.0, COBOL/2 Version 1.0 or FORTRAN/2 Version 1.02.

Dialog Manager Application

Write a Dialog Manager application if you require the following:

- Productive development environment for text-oriented applications
- Compatibility with Presentation Manager graphic interfaces
- Enforcement of Common User Access guidelines.

The *Dialog Manager* text-based application displays panels and help panels in windows on the screen. A tagging language with syntax similar to the International Standards Organization Standard Generalized Markup Language (SGML) is used to define panels and panel elements. A panel tag, for example, defines the panel name, size, and title.

© Copyright IBM Corp. 1989

Other tags define the text and fields contained in the panel. A dialog tag language compiler creates resource files from source tag language files. The resource files are compiled into object code using the Resource Compiler.

The user interface of a Dialog Manager application *automatically* conforms to guidelines in the Common User Access. If you wish to use the graphic interface services of the Presentation Manager, a set of user controls allow you to incorporate graphic elements of the Presentation Manager. This programming activity, however, requires knowledge of the C programming language and the message processing and window procedure concepts necessary to write Presentation Manager applications.

Dialog Manager applications can be written in C/2 Version 1.1, FORTRAN/2 Version 1.02, Macro Assembler/2 Version 1.0, and COBOL/2 Version 1.0.

Text Window Application

The *text window* application does not explicitly create a window in which to run but is supplied with a standard window, which can be moved and sized, by the Presentation Manager. Because the text window application shares the screen with Presentation Manager applications, it must use a subset of the API calls compatible with the Presentation Manager. Generally, the text window application cannot address the hardware directly, and restrictions apply in use of the keyboard, video, and mouse subsystems. The restricted API calls are described in the *Programming Guide*.

The text window environment is well-suited to text-based applications, such as compilers or editors that can operate in the background while a user interacts with another application in the foreground.

The text window application runs in the protected mode of the processor and can use its facilities, such as large memory addressability, I/O protection, multitasking, and dynamic linking.

Full-Screen OS/2 Application

The *full-screen OS/2* application, which is comparable to an application written to run on OS/2 Version 1.0, does not share the screen with other applications. It uses a subset of API calls that is incompatible with the Presentation Manager environment. The full-screen OS/2 application runs in the protected mode of the processor and can use its facilities, such as large memory addressability, I/O protection, multitasking, and dynamic linking.

Full-Screen DOS Application

Existing full-screen DOS applications written to run in the DOS Version 4.0 environment are compatible with OS/2 but cannot share the screen with other applications. Users can start DOS applications from the OS/2 Desktop Manager facility or File Manager facility. Processing of DOS applications is suspended when the application is switched to the background. Therefore, communications, network-dependent, and real-time-dependent applications are not suitable for running in the DOS environment of OS/2. DOS applications are limited to the facilities available in the DOS environment.

Family Applications

The *family application* can execute in the OS/2 and DOS environments of OS/2 and in the DOS 4.0 environment. The API calls used in family applications must be part of the family subset described in the *Programming Guide* and labeled FAPI in the *Control Program Programming Reference*. Generally, the family API calls support applications written to run on DOS 3.3, DOS 4.0, OS/2 Version 1.1, and OS/2 Version 1.2. To build a family application, you must bind the OS/2 executable file, using the BIND utility, to execute in the DOS environment. The API.LIB file, described in the *Building Programs*, maps family API calls used in the application to allow them to execute in the DOS environment.

Comparing the Five Types of Applications

In the present OS/2 environment, there are 16 full-screen sessions, 16 text window sessions, and 16 Presentation Manager sessions. OS/2 uses four full-screen sessions. The Presentation Manager uses two sessions for the spooler and the Program Starter.

A comparison of the features available to each of the five types of applications follows:

	Presentation Manager	**Dialog Manager**	**Text Window**	**Full-Screen OS/2**	**Full-Screen DOS**
Processor mode	Protected	Protected	Protected	Protected	Real
Supported processor	80286/386	80286/386	80286/386	80286/386	8088/286/386
Permits multi-tasking	Yes	Yes	Yes	Yes	No
Runs in back-ground	Yes	Yes	Yes	Yes	No
Intercepts hardware interrupts	No	No	No	No	Yes
Intercepts software interrupts	No	No	No	No	Yes
Obeys 80286 segment rules	Yes	Yes	Yes	Yes	No
Supports memory over-commitment	Yes	Yes	Yes	Yes	No
Addressable memory	16MB	16MB	16MB	16MB	640K
Application residence	<640K/>1MB	<640K/>1MB	<640K/>1MB	<640K/>1MB	<640K

Table 5-1. A Comparison: The Five Types of Applications

Programming Language Support

OS/2 supports the following programming languages:

- C/2 Version 1.1
- Macro Assembler/2 Version 1.0
- COBOL/2 Version 1.0
- FORTRAN/2 Version 1.02
- Pascal Compiler/2 Version 1.0
- BASIC Compiler/2 Version 1.0.

OS/2 does not provide the same level of support to all programming languages. You should be aware of the areas in which the languages are not fully compatible with the Presentation Manager. Only C/2 Version 1.1 and Macro Assembler/2 Version 1.0 provide full support for multi-thread applications and dynamic link libraries.

Two new C/2 run-time libraries support creation of dynamic link libraries. A single-thread application that calls subroutines in dynamic link libraries can link to the LLIBDLL.LIB run-time library. A multi-thread application can link to the CDLLOBJS.LIB run-time library.

Earlier C/2 run-time libraries were not reentrant, meaning that two threads could not use them at the same time; a second thread could overwrite data in the first thread. The supported version of C/2 provides a reentrant version of the standard run-time library, LLIBCMT.LIB, which serves the needs of a multi-thread application. Alternately, a multi-thread application can control access to C/2 run-time libraries by using a semaphore. Note, however, that not all functions in the C/2 run-time libraries are reentrant. The C/2 manual contains a list of all the libraries that are reentrant.

Subroutines you have created and put into dynamic link libraries should not contain static or global variables if you want the subroutines to be reentrant. If the subroutines contain such variables, the application must maintain separate static and global variables for each thread identifier that accesses the subroutine.

COBOL/2 Version 1.0 and FORTRAN/2 Version 1.02, do not support reentrant code. This results in the application creating windows, such as client windows and dialog boxes, that are not system-provided windows. OS/2 usually calls a window procedure by sending the window a message. In COBOL/2 and FORTRAN/2, messages sent to the application's own windows are intercepted by a language-support window procedure. You can get around this limitation by writing a mixed-language application, where the main program is in COBOL/2 or FORTRAN/2, but the window procedures are in C/2 or Macro Assembler/2.

Additionally, COBOL/2 and FORTRAN/2 do not fully support dynamic data exchange, heap management (memory for windows), clipboarding, and window subclassing. The *Programming Guide* provides further details of the limitations of a Presentation Manager window's procedure in COBOL/2 and FORTRAN/2.

Applications written in all supported programming languages must use the Pascal calling convention to access the OS/2 services. The supported compilers (and the assembler) automatically use the Pascal calling convention if the application includes the header files, OS2.H or OS2.INC, provided with the sample programs in the tools package. If the application does not include the header files, the compilers default to the C calling convention, which causes the application to fail. Including the header files is recommended, but you can set the Pascal compiler option or write your own function prototype, declaring each function as Pascal.

The following table summarizes which languages are suitable for generating the five types of applications.

	Presentation Manager	Dialog Manager	Text Window	Full-Screen OS/2	Full-Screen DOS
C/2 Version 1.1	Yes	Yes	Yes	Yes	Yes
Macro Assembler/2 Version 1.0	Yes	Yes	Yes	Yes	Yes
COBOL/2 Version 1.0	Yes	Yes	Yes	Yes	Yes
FORTRAN/2 Version 1.0	Yes	Yes	Yes	Yes	Yes
Pascal Compiler/2 Version 1.0	No	No	No	Yes	Yes
BASIC Compiler/2 Version 1.0	No	No	No	Yes	Yes

Table 5-2. Supported programming languages

National Language Support

If you intend to produce versions of your application in national languages other than American English, 23 national languages with appropriate keyboards and individual country formats for time, date, and currency are supported. Conversion tables for upper- and lower-case characters and collating sequences for sorting characters are provided. API calls are provided to access the services.

Appendix A. Programming Tools and Information Installation

Your tools package provides the following diskettes:

- OS/2 Programming Tools and Information 1
- OS/2 Programming Tools and Information 2
- OS/2 Programming Tools and Information 3
- OS/2 Programming Tools and Information 4
- OS/2 Programming Tools and Information 5
- OS/2 Programming Tools and Information 6.

This appendix will guide you through the installation of these diskettes. A copy of this appendix can also be found as a separate card.

Installation Procedure

1. To **run** the sample programs, you need to install the languages associated with those programs.

To Run:	Install:
C Sample Programs	IBM C/2™[1] Version 1.1
MASM Sample Programs	IBM Macro Assembler/2™[1] Version 1.0
FORTRAN Sample Programs	IBM FORTRAN/2™[1] Version 1.02
COBOL Sample Programs	IBM COBOL/2™[1] Version 1.0

Note:

- The amount of memory used by the C sample programs both in data and lines of code varies. To run all the C sample programs you need to select all memory models when installing IBM C/2, Version 1.1.
- If you use an operating system **other** than OS/2, you must use the UNPACK command to copy files to your fixed disk. The UNPACK command is located on diskette 1 in the \BIN subdirectory. See "UNPACK Command" on page A-3 for information on the UNPACK command.

2. Select **OS/2 Full Screen Command Prompt** from the Group-Main menu.
3. Insert OS/2 Programming Tools and Information 1 diskette in drive A.
4. Type the following at the command prompt:

   ```
   instaid a:toolkit.pip
   ```

5. Press Enter and wait for the IBM Logo to appear on the screen.

 The OS/2 programming tools installation procedure consists of a series of menus and prompts. Your responses to the prompts will move you through the installation.

 Note:

 - You do not have to install all the options now. You may install the programming tools any time you want to add or change options.
 - When all options are installed the Programming Tools and Information use approximately 22MB of disk space.

6. Follow the screen prompts to complete installation of the OS/2 programming tools.

[1] Trademark of IBM Corporation

© Copyright IBM Corp. 1989

The OS/2 Programming Tools Subdirectory Structure

The subdirectory TOOLKT12 is created automatically when you install the tools on your fixed disk. TOOLKT12 contains all subdirectories and files.

The following is the subdirectory structure for the programming tools:

```
—TOOLKT12
    ├—BIN
    ├—C
    │  ├—INCLUDE
    │  └—SAMPLES
    │     ├—BSE
    │     │  ├—CRERRCB
    │     │  ├—DLCBAT
    │     │  ├—EALIST
    │     │  ├—FAMAPIC
    │     │  ├—HELLOCB
    │     │  ├—MEMORYCB
    │     │  ├—MOUSECB
    │     │  ├—VIOCBAT
    │     │  └—WTCBAT
    │     ├—DM
    │     │  └—DMDEMO
    │     └—PM
    │        ├—AVIOSAMP
    │        ├—BMAP
    │        ├—CLIPBRD
    │        ├—DIALOG1
    │        ├—DIALOG2
    │        ├—FONTTEST
    │        ├—GRAPHIC1
    │        ├—GRAPHIC2
    │        ├—HELLO1
    │        ├—HELLO2
    │        ├—HELP1
    │        ├—IMAGE
    │        ├—TEMPLATE
    │        └—TYPETEXT
    ├—COBOL
    │  ├—INCLUDE
    │  └—SAMPLES
    │     ├—DM
    │     │  └—DMDEMO
    │     └—PM
    │        ├—DLG1COB
    │        └—DLG2COB
    │
    │
```

The subdirectory structure is continued on the next column.

Programming Tools Subdirectory Structure Continued:

```
    │
    ├—DLL
    ├—DM
    ├—DTL
    ├—FORTRAN
    │  ├—INCLUDE
    │  └—SAMPLES
    │     └—PM
    │        ├—DLG1FOR
    │        └—DLG2FOR
    ├—IPFC
    ├—LIB
    ├—MASM
    │  ├—INCLUDE
    │  └—SAMPLES
    │     ├—BSE
    │     │  ├—CRERRB
    │     │  ├—DEMODB
    │     │  ├—DYNBLD
    │     │  ├—FAMAPI
    │     │  ├—HELLOBAT
    │     │  ├—IOPL2B
    │     │  ├—MEMORYB
    │     │  ├—MOUSAMPB
    │     │  ├—PROCESSB
    │     │  ├—PROMODE
    │     │  ├—THREADB
    │     │  ├—VIOSAMPB
    │     │  └—WTBAT
    │     └—PM
    │        └—HELLOM
    ├—PD
    │  └—SAMPLES
    │     └—PMPRT
    ├—PROGREF
    └—REXX
       └—SAMPLES
          ├—CALLREXX
          ├—PMREXX
          ├—RXMACDLL
          ├—RXSTRING
          └—RXSUBDLL
```

Notes:

1. The subdirectory structure on your fixed disk may differ, depending on the options you install.
2. See the **readme.c** file in the TOOLKT12\C subdirectory, **readme.for** file in the TOOLKT12\FORTRAN subdirectory, **readme.cbl** file in the TOOLKT12\COBOL subdirectory, and **readme.asm** file in the TOOLKT12\MASM subdirectory for descriptions of the subdirectory files.

UNPACK Command

Some of the files on the diskettes are compressed. When the tools diskettes are installed under OS/2 the files are unpacked automatically. When the tools diskettes are installed under any other operating system the files must be unpacked manually.

Files that are "packed" can be recognized by the @ symbol acting as the third character in the file name extension. Those extensions with only one character have blanks padded with underscores (_); for example:

Original	Compressed
FILE1.COM	FILE1.CO@
FILE2.H	FILE2.H_@

Do not specify an output filename; UNPACK uses the original uncompressed filename and extension as the destination filename. It also preserves the date, time and file attribute of the original uncompressed file from the header of the compressed file.

UNPACK copies files that are not compressed and handles file information such as date, time, and file attributes in the same way the COPY command does. Therefore, there can be a combination of compressed and non-compressed files on a diskette.

If You Want To:	Enter:
Copy a compressed file from the current directory on drive A to drive C	`A:\>UNPACK A:FILE.CO@ C:`
Copy an entire diskette of compressed and uncompressed files to the root directory on drive C	`A:\>UNPACK A:\*.* C:`

Index

A

accelerator tables 3-8
addressing memory
 Global Descriptor Table 2-2
 Local Descriptor Table 2-2
 real addressing mode 2-1
 segment descriptors 2-2
 segment tables 2-2
 selectors 2-2
 virtual addressing mode 2-1
all-points-addressable device 3-7
allocating memory 2-11
alphanumeric output 3-10
 direct to a device 3-11
 using a spool queue 3-10
 using Vio calls 3-10
APA, all-points-addressable device 3-7
API (see Application Programming Interface)
Application Programming Interface (API)
 advantages 2-3
 calls, prefixes 2-2
 Dev prefix 2-2
 Dos prefix 2-2
 extending 2-3
 Gpi prefix 2-2
 implementing 2-3
 Kbd prefix 2-2
 Mou prefix 2-3
 responsibilities of 2-3
 Spl prefix 2-3
 Vio prefix 2-3
 Win prefix 2-3
application queue 3-5
application resources 3-7
 accelerator tables 3-8
 bit maps 3-7
 compiling 3-8
 creating using editors 3-8
 definition 3-7
 dialog and window templates 3-8
 dialog includes 3-7
 editing 3-8
 fonts 3-7
 icons 3-7
 menu 3-8
 pointers 3-7
 string tables 3-7
 types of 3-7
application structure 3-3
application types 5-1
 comparison of 5-3
 Dialog Manager 5-1
 family 5-2
application types *(continued)*
 full-screen DOS 5-2
 full-screen OS/2 5-2
 Presentation Manager 5-1
 text window 5-2
assembling a graphics picture 3-9
assignment lists, Dialog Manager 4-4
AVIOSAMP 1-8

B

BASIC Compiler/2 1-2
BIND 1-4
bindings references 1-3
bit maps 3-7, 3-10
block device 2-9
BMAP 1-8

C

cached presentation space 3-9
changing a graphic object 3-9
character device 2-9
character graphics 3-8
check lists, Dialog Manager 4-4
child process 2-4
child window 3-3
client area 3-6
client window 3-6
clipboard
 cut, copy, paste metaphor 3-11
 definition 3-11
 formats 3-11
 ownership 3-11
CLIPBRD 1-8
clipping 3-3, 3-9
COBOL/2 1-2, 5-4
CodeView utility 1-5
command tables 4-3
Common Programming Interface (CPI) 3-1
Common Programming Interface (CPI)
Common User Access 3-1
 dialog box window 3-5
 Dialog Manager applications 4-5
 message window 3-5
 primary window 3-5
 secondary window 3-5
 supplemental window 3-5
comparison of types of applications 5-3
compilers
 Dialog Tag Language compiler 4-3
 Information Presentation Facility Compiler 3-7
compiling sample programs 1-5

© Copyright IBM Corp. 1989

complex graphics 3-9
concurrent processing 2-3
context switches 2-12
control services, Dialog Manager 4-3
control window 3-6
CPI, Common Programming Interface 3-1
creating graphics 3-8
CRERRCB.C 1-7
cursor 3-4
C/2 1-2, 5-4

D

data exchange
 application-generated 3-12
 user-generated 3-11
DEMODB.ASM 1-7
descriptors 2-2
desktop window 3-3
Dev calls 2-2
DevHlps 2-10
device contexts
 and presentation space 3-10
 definition 3-10
device driver 2-9
device handle 2-8
device helper services 2-10
device I/O
 and file I/O 2-8
device management 2-9
device monitor 2-10
device presentation sample program, printer 1-4
device space 3-9
devices
 and device monitor 2-10
 block 2-9
 character 2-9
 DevHlp services 2-10
 I/O control 2-10
 I/O privilege 2-10
 standard 2-8
dialog and window templates 3-8
dialog box 3-6
 and message box 3-6
 definition 3-6
 dialog procedure 3-6
 generating 3-6
 includes 3-7
 modal 3-6
 modeless 3-6
 template 3-6
Dialog Box Editor 1-4, 3-8
dialog box window 3-5
Dialog Manager
 assignment lists 4-4
 command tables 4-3
 Common User Access 4-5
 compiler for tag language 4-3
Dialog Manager *(continued)*
 difference in a Presentation Manager application 4-6
 display services 4-3
 DMDEMO sample program 4-5
 double-byte character support 4-8
 help interface 4-6
 Information Presentation Facility 4-6
 library services 4-3
 messages 4-3
 national language support 4-7
 panel elements 4-1
 single-byte coded characters 4-8
 translate lists 4-4
Dialog Manager Application 5-1
dialog tag language 4-1
Dialog Tag Language Compiler 1-4, 4-3
dialog variable pool 4-4
dialog variables 4-4
DIALOG1 1-8
DIALOG2 1-8
directory 2-8
diskettes in tools package A-1
display services for Dialog Manager 4-3
DLNK1C.C 1-7
DLNK2C.C 1-7
DLNK3C.C 1-7
DMDEMO sample program 4-5
DOS application, full-screen 5-2
DOS calls 2-2
double-byte character set support 4-8
dynamic link libraries
 building 2-11
 definition 2-11
dynamic linking
 imported libraries 2-11
 load-time 2-10
 run-time 2-11
DYNLINKC.C 1-7

E

exit list 2-4
exported subroutines 2-11
extended attributes 2-8

F

family application 5-2
fast-safe RAM semaphore 2-7
file
 handle 2-8
 I/O 2-7
 locking 2-8
file system
 directories 2-8
 file 2-7
 hierarchy 2-7

file system *(continued)*
high performance file system 2-7
long file names 2-7
root directory 2-8
sub-directories 2-8
filtering input 3-5
fixed-high thread 2-5
flags 2-5
Font Editor 1-4, 3-8
fonts 3-7
FONTTEST 1-8
FORTRAN/2 1-2, 5-4
frame window 3-6
full-screen DOS application 5-2
full-screen OS/2 application 5-2

G

GDT, Global Descriptor Table 2-2
geometric coordinates 3-9
Global Descriptor Table (GDT) 2-2
Gpi calls 2-2
Gpi prefix 3-8
graphic objects 3-8
graphic output 3-9
direct to a device 3-11
handling 3-9
presentation space 3-9
using the spooler 3-10
graphic segment 3-8
graphics
character 3-8
complex 3-9
coordinates 3-9
creating 3-8
managing 3-8
primitives 3-8
segment chain 3-8
segment store 3-8
storing 3-8
types of 3-8
graphics coordinate spaces
and clipping 3-9
device space 3-9
model space 3-9
presentation space 3-9
world coordinate space 3-9
GRAPHIC1 1-8
GRAPHIC2 1-8

H

handle-based devices 2-8
handling input 3-3
hard-copy output 3-10
alphanumeric 3-10
graphic 3-10
producing 3-10
hardware systems 2-1
header files 1-5
COBOL/2 1-5
C/2 1-5
FORTRAN/2 1-5
Macro Assembler/2 1-5
HELLOCB.C 1-7
HELLOM 1-8
HELLO1 1-8
HELLO2 1-8
help interface
for Dialog Manager applications 4-6
for Presentation Manager 3-7
Information Presentation Facility 3-7
help panels 3-7
high performance file system 2-7
hot spot on mouse 3-4

I

Icon Editor 3-8
icons 3-7
idle-time thread 2-5
IMAGE 1-9
IMPLIB utility 1-4
import library 2-11
include files 1-5
Information Presentation Facility 3-7
initializer routine for dynamic linking 1-7
input handling 3-3
installation procedure A-1
installing the Programming Reference 1-2
installing the tools 1-4
inter-process communication facilities
definition 2-5
flags 2-5
messages 2-6
pipes 2-5
queues 2-6
semaphores 2-6
shared memory 2-6
inter-process communication (IPC) 2-5
interactive applications 4-1
IOCtl, see I/O control 2-10
IOPL2B.ASM 1-8
IPC, inter-process communication 2-5
I/O control 2-10
I/O for handle-based devices 2-8
I/O privilege (IOPL) 2-10
I/O subsystems 2-8

K

Kbd application interfaces 2-9
key mapping lists 4-3
keyboard input 3-4
keyboard subsystem 2-8

L

language limitations 5-4
language support, national 5-5
LDT, Local Descriptor Table 2-2
library services, Dialog Manager 4-3
LINK utility 1-4
load-time dynamic linking 2-10
loader 2-10
Local Descriptor Table (LDT) 2-2

M

Macro Assembler/2 1-2, 5-4
main windows 3-3
MAKE utility 1-5
managing devices 2-9
managing memory 2-11
markup language for help panels 3-7
memory 2-11
 allocating 2-11
 application interfaces 2-11
 deallocating 2-11
 managing 2-11
 overcommitment 2-1
 segment swapping 2-1
 suballocating 2-11
 80386 extended registers 2-12
memory protection
 descriptors 2-2
 I/O privilege 2-10
 OS/2 protection model 2-2
 privilege levels 2-2
 privilege model 2-10
memory suballocation 2-11
MEMORYCB.C 1-7
menu resource 3-8
message box 3-6
message processing 3-4
message window 3-5
messages 2-6
metafile 3-10
micro presentation space 3-9
MKMSGF 2-12
MKMSGF utility 1-4
modal dialog box 3-6
model space 3-9
modeless dialog box 3-6
module definition file 2-10
MONITOR.ASM 1-7
Mou application interfaces 2-9
Mou calls 2-3
mouse capture window 3-4
mouse pointer 3-4
mouse subsystem 2-8
MOUSEC.C 1-7
MSGBIND 2-12
MSGBIND utility 1-5
multitasking 2-3
 advantages 2-3
 definition 2-3
 hierarchy 2-4
 processes 2-4
 sessions 2-4
 threads 2-4
multitasking hierarchy 2-4

N

named pipe 2-6
named shared memory 2-6
national language support 5-5
national language support, Dialog Manager 4-7
normal presentation space 3-9

O

online help panels 3-7
online Programming Reference installation 1-2
OS2STUB.EXE 1-4
OS/2 operating environments 2-1
OS/2 programming concepts 2-1
OS/2 text messages
 definition 2-12
 displaying to user 2-12
 necessary utilities 2-12
owner window 3-3

P

parent process 2-4
parent window 3-3
Pascal Compiler/2 1-2
pipes 2-5
pointers 3-7
presentation drivers
 compared to OS/2 device drivers 3-12
 definition 3-12
Presentation Manager application 5-1
Presentation Manager concepts 3-1
Presentation Manager messages
 and window procedures 3-5
 application queue 3-5
 filtering input 3-5
 input events 3-4
 processing 3-4
 system queue 3-4
presentation spaces
 alphanumeric 3-10
 and device context 3-10
 cached 3-9
 definition 3-9
 graphics 3-9
 micro 3-9
 normal 3-9

primary window 3-5
primitives 3-8
printer device driver sample program 1-4
printing 3-10
private window class 3-2
Procedures Language 1-2
process switches 2-12
processes
 and exit lists 2-4
 child 2-4
 definition 2-4
 in multitasking hierarchy 2-4
 parent 2-4
processor modes
 protected 2-1
 real 2-1
PROCESS.ASM 1-7
programming aids 1-4
programming languages 1-2
 BASIC Compiler/2 1-2
 COBOL/2 1-2
 C/2 1-2
 FORTRAN/2 1-2
 levels of support 5-4
 Macro Assembler/2 1-2
 Pascal Compiler/2 1-2
 Procedures Language 1-2
 supported 1-2, 5-4
Programming Reference, online 1-2
PROG1.ASM 1-7
PROG2.ASM 1-7
PROMODE.ASM 1-8
protected addressing mode 2-1
protected mode 2-1
public window class 3-2

Q

queues 2-6

R

RAM semaphore 2-7
RC, Resource Compiler 1-4
real addressing mode 2-1
real mode 2-1
REALMODE.ASM 1-8
rectangle 3-2
regular thread 2-5
replacing a subsystem 2-9
Resource Compiler 1-4, 3-8
Resource Compiler (RC) 1-4
Resource Editor 3-8
resource editors 1-4
 dialog box 1-4, 3-8
 font 3-8
 Font Editor 1-4
 icon 3-8
resource editors *(continued)*
 Icon Editor 1-4
resources 3-7
root directory 2-8
rotating graphics objects 3-9
run-time dynamic linking 2-11

S

SAA, Systems Application Architecture 3-1
sample programs 1-4, 1-5, 1-7
 AVIOSAMP 1-8
 BMAP 1-8
 CLIPBRD 1-8
 compiling and running 1-5
 CRERRCB.C 1-7
 DEMODB.ASM 1-7
 DIALOG1 1-8
 DIALOG2 1-8
 DLG1COB.CBL 1-8
 DLG1FOR.FOR 1-8
 DLG2COB.CBL 1-8
 DLG2FOR.FOR 1-8
 DLNK1C.C 1-7
 DLNK2C.C 1-7
 DLNK3C.C 1-7
 DMDEMO 1-9
 DYNLINKC.C 1-7
 D_INITC.C 1-7
 EALIST 1-8
 FONTTEST 1-8
 GRAPHIC1 1-8
 GRAPHIC2 1-8
 HELLOCB.C 1-7
 HELLOM 1-8
 HELLO1 1-8
 HELLO2 1-8
 IMAGE 1-9
 IOPL2B.ASM 1-8
 MEMORYCB.C 1-7
 MONITOR.ASM 1-7
 MOUSEC.C 1-7
 PMPRT 1-9
 PROCESS.ASM 1-7
 PROG1.ASM 1-7
 PROG2.ASM 1-7
 PROMODE.ASM 1-8
 REALMODE.ASM 1-8
 required C/2 libraries 1-5
 required C/2 options 1-5
 TEMPLATE 1-9
 THREADB.ASM 1-7
 TYPETEXT 1-9
 VIOSAMPC.C 1-7
 WTC.C 1-7
scaling graphics objects 3-9
secondary window 3-5

segment by name, accessing 2-6
segment chain 3-8
segment registers, 80386 2-12
segment store 3-8
segment swapping 2-1
segment tables, types of 2-2
selectors 2-2
semaphores
 definition 2-6
 fast-safe RAM 2-7
 RAM semaphore 2-7
 system semaphore 2-7
sessions
 definition 2-4
 examples 2-4
 in multitasking hierarchy 2-4
shared memory 2-6
shearing graphics objects 3-9
sibling window 3-3
source header files 1-5
source include files 1-5
Spl calls 2-3
spooler 3-12
standard devices 2-8
Standard Generalized Mark-up Language 4-3
standard window 3-5
 client area 3-6
 controls 3-6
 definition 3-5
 elements 3-5
 frame 3-6
standard window elements 3-5
starting a DOS application 5-2
storing graphics 3-8
string tables 3-7
structuring the application 3-3
sub-directory 2-8
suballocating memory 2-11
subsystems
 and sessions 2-9
 Kbd interfaces 2-9
 keyboard 2-9
 Mou interfaces 2-9
 replacing 2-9
 video 2-9
 Vio interfaces 2-9
supplemental window 3-5
system configuration 2-1
system limits 2-12
system queue 3-4
system semaphore 2-7
system variables, Dialog Manager 4-4
Systems Application Architecture (SAA) 3-1
Systems Application Architecture (SAA)
 and Presentation Manager 1-3, 3-1
 Common Programming Interface (CPI) 3-1
 Common User Access 3-1
 supported hardware 3-1
Systems Application Architecture (SAA) *(continued)*
 supported operating environments 3-1

T

tag language, dialog 4-1
TEMPLATE 1-9
text window application 5-2
thread priority classes
 fixed-high 2-5
 idle-time 2-5
 regular 2-5
 time-critical 2-5
thread switches 2-12
THREADB.ASM 1-7
threads
 definition 2-4
 dispatching priority 2-5
 in multitasking hierarchy 2-4
time-critical thread 2-5
time-slicing 2-5
tools installation 1-4
transforming pictures
 definition 3-9
 graphic objects 3-9
 rotating 3-9
 scaling 3-9
 shearing 3-9
 translating 3-9
translate lists, Dialog Manager 4-4
translating graphics objects 3-9
TYPETEXT 1-9

U

UNPACK command A-3
user input
 cursor movement 3-4
 handling 3-3
 keyboard 3-4
 mouse 3-4
user interface
 and Common User Access guidelines 3-5
 providing 3-5
utilities 1-4
 BIND 1-4
 CodeView 1-5
 Dialog Tag Language Compiler 1-4
 IMPLIB 1-4
 LINK 1-4
 MKMSGF 1-4, 2-12
 MSGBIND 1-5, 2-12
 RC 1-4
 Resource Compiler 1-4
 WINABLE 1-5

V

variable class 4-2
variable pool 4-4
variable services, Dialog Manager 4-3
video subsystem 2-8
Vio application interfaces 2-9
Vio calls 2-3
Vio prefix 3-8
VIOSAMPC.C 1-7
virtual addressing mode 2-1
virtual memory 2-1
virtual mode 2-1

W

Win calls 2-3
WINABLE utility 1-5
window class 3-2
window handle 3-2
window hierarchy 3-2
window procedure 3-2
window relationships 3-2
window words area 3-2
windowing environment 3-2
windows
 appearance 3-2
 as rectangles 3-2
 behavior 3-2
 child 3-3
 classes 3-2
 classifying 3-2
 client 3-6
 clipping 3-3
 control 3-6
 coordinates for sizing 3-2
 creating 3-2
 definition 3-2
 desktop 3-2
 dialog box, definition of 3-5
 elements 3-5
 frame 3-6
 hierarchy 3-2
 identifiers 3-2
 main 3-3
 mouse capture 3-4
 owner 3-3
 parent 3-3
 positioning 3-2
 primary, definition of 3-5
 private classes 3-2
 public classes 3-2
 relationships 3-2
 secondary, definition of 3-5
 sibling 3-3
 sizing 3-2
 standard 3-5
 sub-classifying 3-2
windows *(continued)*
 supplemental, definition of 3-5
 window words area 3-2
 z – order 3-3
world coordinate space 3-9
WTC.C 1-7

Numerics

16 bit 80386 segment registers 2-12
80386 extended registers 2-12

IBM United Kingdom
International Products Limited
PO Box 41, North Harbour
Portsmouth, PO6 3AU
England

Printed in Denmark by Rosendahl · Esbjerg

